AFHRL-TR-90-45

(2)

AIR FORCE

HUMAN RESOURCES

DEBUGGING TECHNIQUES USED BY EXPERIENCED PROGRAMMERS TO DEBUG THEIR OWN CODE

Pamela M. Merrick, Captain, USAF

OPERATIONS TRAINING DIVISION
Williams Air Force Base, Arizona 85240-6457

September 1990

Final Technical Report for Period July 1989 - April 1990

LABORATORY

AIR FORCE SYSTEMS COMMAND
BROOKS AIR FORCE BASE, TEXAS 78235-5601

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The Public Affairs Office has reviewed this report, and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This report has been reviewed and is approved for publication.

DEE H. ANDREWS, Technical Director
Operations Training Division

HAROLD G. JENSEN, Colonel, USAF
Commander

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1990	Final Report - July 1989 to April 1990

4. TITLE AND SUBTITLE
Debugging Techniques Used by Experienced Programmers to Debug Their Own Code

5. FUNDING NUMBERS
PE - 62205F
PR - 1123
TA - 05
WU - 01

6. AUTHOR(S)
Pamela M. Merrick

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Operations Training Division
Air Force Human Resources Laboratory
Williams Air Force Base, Arizona 85240-6457

8. PERFORMING ORGANIZATION REPORT NUMBER
AFHRL-TR-90-45

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The present research examines professional programmers' attitudes toward interactive debuggers and studies the techniques they use in debugging their own code. Professional programmers were asked to fill out a questionnaire regarding their use and evaluation of debuggers. The programmers were then asked to code and debug three progamming tasks. Protocol data, videotapes, and intermediate versions of the source code were used to analyze debugging techniques. The results suggest that available debuggers meet programmers' functional requirements, but the presentation of the debuggers needs to be improved. Implications for future debugger development are discussed.

14. SUBJECT TERMS
code
computer
debug
debugging
programmers
programming

15. NUMBER OF PAGES
62

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

SUMMARY

Mistakes in software programs are isolated and repaired by a process called "debugging." Programmers can use software debugging tools to help them find and fix errors in their code more rapidly than debugging by hand. Less than one-third of programmers, however, use debugging tools. Programmers may not be using them because the tools do not allow them to use their standard techniques. The present research investigated the techniques used by programmers as they debug their own code. By learning how programmers typically debug code, software debugging tools can be designed to take advantage of programmers' established modes of operation.

Twelve programmers served in this experiment. They completed a questionnaire and were given three programming tasks to code. The subjects' coded data were examined to find the errors that had to be debugged. Each error was identified and the technique the subject used to find the error was noted.

Overall, the programmers used the code, the output, debug print statements, and hand-simulation to find their bugs. The programmers' debugging techniques, insertion of debug print statements and hand-simulation are supported by most debuggers; and the results would have been more accurate, in that the system would not have overlooked any line of code. None of the subjects in this experiment, however, used the debugger.

The results of this experiment imply that debuggers would be used more frequently if they were easier to learn. Because the programmers' functional needs are being met, designers should now concentrate on the presentation of the debugger, including interface design, documentation, and marketing.

PREFACE

The general objective of the research described in this report was to analyze methods used by professional programmers in debugging their own code. The results were used to evaluate the functional correspondence between software debugging tools and current debugging methods.

This effort supports the Training Technology objective of the Air Force Human Resources Laboratory (AFHRL) Research and Technology Plan by identifying and demonstrating cost-effective ways of developing and maintaining new skills.

This work was accomplished at the Air Force Human Resources Laboratory's Operations Training Division (AFHRL/OT) and performed by Captain Pamela M. Merrick, Principal Invesigator, under Work Unit 1123-05-01, In-House Research and Development Support. This work would not have been possible without the oustanding support of 2Lt Paul Weiss and 2Lt Joe Drbohlav from the 82 Flying Training Wing, who were instrumental in preparing the protocol data for analysis. The author also wishes to thank Dr. Peter M. Crane, AFHRL/OT, who provided invaluable critiques and comments throughout the duration of this effort.

TABLE OF CONTENTS

LIST OF TABLES

DEBUGGING TECHNIQUES USED BY EXPERIENCED PROGRAMMERS TO DEBUG THEIR OWN CODE

I. INTRODUCTION

Software programs seldom work correctly when first coded. If a program does not meet its specifications after having been coded, mistakes are isolated and repaired by debugging. Debugging is generally considered to be the most expensive phase of software production (Sheil, 1981), due to the extensive testing required to find and fix all errors. Between 40% and 70% of a programmer's time is spent removing "bugs" (Seviora, 1987). The cost to fix a mistake increases as the development cycle progresses. At Digital Equipment Corporation, it has been estimated (Harris, 1988) that if software bugs cost $1 to fix before coding, the relative cost to fix software bugs is $1.50 during coding, $10 before field testing, $60 during field testing, and $100 in the field. Therefore, debugging code early in the programming process minimizes production costs.

Aside from the costs involved, debugging is also a frustration of creative talents. Once a programmer has coded his work, he wants to get on to the next problem; so, the debugging is done in an atmosphere of impatience (Brown & Sampson, 1973). This general lack of enthusiasm for debugging is heightened when a programmer has to debug someone else's code (Seviora, 1987).

Software debugging tools, such as VAX DEBUG (Beander, 1983), can help programmers find and fix errors in their code more rapidly than debugging by hand. However, in a study of software engineering practices in 30 companies, Zelkowitz, Yeh, Hamelt, Gannon, and Basili (1984) found that only 27% of the programmers used any kind of testing tools. Many had debuggers available, but were not using them. Simple code reading was the preferred approach. The reasons given for lack of use ranged from "hard to learn" to "not very useful." It is possible that the majority of programmers do not use debugging tools because the tools do not reflect the debugging techniques they most typically employ. By learning how expert programmers typically debug code, more useful software tools could be designed.

Most studies on debugging practices have concentrated on novice programmers and fail to offer guidance in developing advanced software

development environments (Curtis, 1986). Shneiderman (1986) has suggested that the design of testing and debugging tools would benefit from human factors studies of how experienced programmers do testing and debugging.

Literature Review

Prior Debugging Studies

Prior research on debugging falls into three categories, according to who is being studied: novice programmers, novices compared to experts, and expert programmers only.

Kessler and Anderson (1986) investigated true novices who had no previous programming experience. After attending a one-day LISP tutorial, the novices attempted to debug 12 one-line LISP functions. The protocol data collected showed that six of the seven subjects used the same four-step debugging strategy for all problems. First they tried to understand what the code was doing. Next, they ran the code to detect the error. Then they searched for the piece of code responsible for the error. Finally, they corrected the bug. The fourth step, bug repair, was found to be extremely difficult and independent of the first three steps.

Carver and Klahr (1986) broke the novice debugging strategy into more detailed steps in an effort to develop debugging instruction. Their strategy was as follows:

1. Test the program by running it.
2. Compare actual output to desired output and evaluate the differences.
3. Describe any discrepancies.
4. Propose possible bugs and ways to find them.
5. Look for information about the program structure's data representation.
6. Specify each bug's likely location within the structure.
7. Search and find the bugs in the code.
8. Interpret each command and check the command's effect.
9. Change code by replacing bad code with the appropriate correction.
10. Begin again by running the program to test it.

Carver and Risinger (1987) found that speed and efficiency improved when children were taught how to narrow the search for bugs using this strategy.

Spohrer and Soloway (1986) examined the types of bugs introduced by novices, as opposed to novice debugging strategies. They analyzed the first syntactically correct versions of 10 programs submitted by 61 students during a semester-long course. Bug type and frequency were examined. They observed that 20% of the bug types accounted for 55% of the errors. Based on analysis of the most common bug types, Spohrer and Soloway concluded that novices had difficulties with the following seven items: deciding appropriate boundary points, detecting dependencies that affect nesting, negation of complex logic, dropping the final digit on constants with repeating tail digits, determining the precedence of operators, assuming incorrect ways to calculate quantities from specification misinterpretation, and interference of similar numbers with incorrect answers.

Several studies have been conducted comparing novice and expert programmers' debugging techniques. Jeffries (1982) had six experts and four novices debug two Pascal programs that contained several bugs each. Vessey (1985) asked eight novices and eight experts to debug a COBOL program with one error in it. Gugerty and Olson (1986) ran two groups of subjects. The first group consisted of eighteen experts and six novices who debugged three LOGO programs with one bug in each program. Ten novices and ten experts comprised the second group for debugging a Pascal program containing one error. Finally, Nanja and Cook (1987) used a Pascal bubble sort program with six errors to test six novices, six intermediates, and six experts.

All five of these studies produced similar results when comparing expert to novice debugging performance. The experts started by reading the code in the order that it would be executed. They spent a fair amount of time gaining a high-level understanding of how the program functioned. Once they understood the program and what it was supposed to do, they applied this knowledge to place the error in context. Experts then quickly located the bug and often corrected it properly the first time, and never introduced new bugs. Experts found more bugs and found them faster. Nanja and Cook's experts (1987) were the only ones who used the interactive debugger. In all five studies, the novices used a much less organized approach. First, they read the code from top to bottom, regardless of actual execution order.

They then immediately jumped in and tried to find the bugs. Due to lack of program comprehension, their initial hypotheses were inferior and made it hard to find relevant parts of the code. Without an overall understanding, it took them longer to confirm or reject error corrections. Novices often added new bugs by forgetting to undo incorrect fixes.

Holt, Boehm-Davis, and Schultz (1987) also compared experts and novices, but focused on the way a computer program is represented cognitively and how that representation is used. The experts' mental models were significantly affected by the difficulty of the module; that is, the more difficult the program, the more elaborate the mental model. The structure and content of the program, however, did not affect the experts' mental models. The novices, on the other hand, were not affected by module difficulty, but were affected by program structure. This finding implies experts are better at abstracting the code and less influenced by the superficial or peripheral aspects of the code. This ability to abstract code was crucial in the programmers' performance. The researchers also found the number of languages and operating systems used, as well as the number of programs written, to be a better indicator of an expert than years of schooling or professional programming.

Three studies examined expert programmers only. Gould and Drongowski (1974) and Gould (1975) explored the techniques used by Fortran programmers to find bugs in 12 programs, each containing a one-line bug. The bugs were categorized as array indexing bugs, iteration loop bugs, and assignment statement bugs. In both studies, it was found that assignment statement bugs were three times harder to find than the other two types and debugging was three times more efficient on programs that had been previously debugged with different bugs. In Gould's (1975) study, an interactive debugger was also made available to the programmers; however, they rarely made use of it. The general strategy used was to select a debugging tactic and search for something suspicious. After a clue was found, a hypothesis about the bug was generated in testable terms from previously acquired knowledge. If the hypothesis was correct, the bug had been found. Otherwise, a new clue was identified and the cycle repeated. Programmers tended to ease into debugging, Gould also found. First they eliminated all syntactic bugs, then grammatical bugs not detected by the compiler, and finally the substantive

bugs. Also, they avoided difficult sections of the code until the rest of the code had been eliminated.

Guidon, Krasner, and Curtis (1987) studied breakdowns that occur while expert programmers design complex software. They used protocol data collected while subjects spent 2 hours designing logic for the N-lift problem. During the design process, the programmers were found to place great emphasis on using mental simulations to understand and elaborate on the requirements.

Breakdowns occurred for several reasons. Some subjects lacked specialized knowledge about design schemes that could be instantiated. Some subjects lacked knowledge about design process goals and alternatives to guide them in the amount of effort to spend on different activities. Poor prioritization of issues and constraints led to poor selection from alternative solutions. Subjects had difficulty considering all the stated or inferred constraints in refining their solutions, due to short-term memory capacity. Subjects had difficulty performing mental simulations with many steps or many test cases, again due to cognitive limitations. Subjects had difficulty keeping track and returning to postponed sub-problems. Finally, it was difficult for subjects to expand and merge their partial solutions into a complete solution.

The Debugging Process

Based on these empirical studies and personal debugging experience, many authors have tried to describe the process of debugging. Knoke (1988) divided the debugging process into four separate steps: testing, stabilization, localization, and correction. During the testing phase, a wide range of input values is used to force execution of all program branches. The program's capabilities are tested, starting with normal values and then proceeding to boundary conditions and special cases. Any anomaly indicates a possible bug. The programmer must rely on his knowledge of what to check and how to analyze the resulting output. The program must then be stabilized so that specific bugs can be generated at will. Tne programmer needs to be in control of the conditions which cause the bug. The bug must then be isolated to a specific variable or segment of code. This localization can be done any of three ways. The programmer can

single-step through the subject code until abnormal behavior is noted. A trace history of the executed code may be examined, or the programmer can hypothesize and modify code to test the hypothesis validity. The bug must then be corrected. After correction, the programmer must begin again with testing. This is done to make sure the edit achieves the intended purpose and has no side effects.

Seviora (1987) pointed out that programmers usually combine Knoke's method with a problem analysis approach. The main emphasis in the problem analysis approach is to eliminate discrepancies between the specification (what the program should do) and the source code (what the program actually does). To do this, the programmer relies on his knowledge of program constructs to understand what the code does and how its parts interact.

Several authors have suggested that one aid to understanding is to reduce the amount of detail by extracting only relevant information. Weiser (1982) ascertained that programmers mentally divide their code into functional sections when debugging. They break apart large programs into smaller coherent pieces, which are not necessarily contiguous parts of the text. Lukey (1980) also found that programmers segment programs into chunks of code that form useful units of analysis. The segments are used to identify pieces of code that are likely to be bugged and eliminate portions of the program that seem to be running correctly. One method used to identify the bad code is backtracing. A path is traced back through the immediately preceding assertions related to the discrepancy (Lukey, 1980). By working backwards from the symptoms, bugs are often easier to locate. Both Gould (1975) and Lukey (1980) noted instances of programmers working backwards from an error's appearance to locate its source.

Rasmussen (1986) found that programmers also use prior debugging experience to recall previous bugs that caused symptoms similiar to the current one. In fact, much of debugging skill is learned through the experience of writing programs and getting them to run (Gugerty & Olson, 1986). The more experienced programmer has a larger mental library of symptom bug associations. Faced with less familiar situations, programmers resort to casual reasoning directly from the source code (Seviora, 1987). When completely clueless, programmers hand-simulate the execution of their

program on paper. They may also add debug print statements and change program statements just to see what will happen (Rasmussen, 1981).

Overview

The present research examines debugging strategies used by experienced programmers debugging their own code. By thoroughly studying the techniques programmers employ, perhaps we can discover why software debuggers are not more widely used. The research focused on isolating the strategies applied, so that future debuggers can be designed around the methods programmers already use.

The technique of protocol analysis was used for data collection and analysis. Each subject was asked to verbalize his thoughts while programming the assigned tasks. The protocol data were coded and used to isolate the errors the programmers made. The analysis attempted to discover the techniques used to correct the bugs.

Six different classes of programming errors have been described in the programming literature. Brown and Sampson (1973) called the first class of errors "appreciation errors." These bugs result from the programmer's misinterpreting a specification or failing to read it thoroughly before starting work. The second class covers lexical and syntactic errors. These include violation of a language's grammar rules, typographical errors and incorrect translation of an algorithm to program code. Since the compiler usually catches these errors, they are usually the easiest to correct. The next two classes are run-time errors called "execution errors" and "intent (logic) errors" (Knoke, 1988). Mistakes are called execution errors when a program terminates abnormally due to run-time checks, out-of-bounds, etc. With intent errors, the program runs, but it produces incorrect results. These can be caused by design flaws or incomplete comprehension of the problem. The final classes of errors are integration and portability errors (Knoke, 1988). Integration problems will not appear until two or more modules are combined to form a program, and portability bugs show up when code is moved from one machine to another.

In most of the previous debugging studies, programmers debugged code that contained a predetermined number of bugs. The bugs were carefully chosen and placed in the code by the experimenters. In doing so, the

experimenters limited the bugs to one or two error classes. The number of error types encountered by programmers debugging their own code, on the other hand, is limited only by the programmer's skill. This research should therefore be able to cover all except the final bug classes (integration and portability).

Protocol Analysis

The technique of protocol analysis was used to examine debugging strategies employed by experienced programmers. Protocol analysis is ideally suited to the development and testing of theories in the emerging computer programming domain (Fisher, 1987). In protocol analysis, subjects are asked to talk/think aloud as they solve problems. The verbalizations are then used to track the cognitive processes used to perform the task. The method has been used to study a variety of complex mental tasks. Rasmussen (1981) used protocol analysis to invest te diagnostic strategies used by technicians to find faults in an operational process plant control. Blackman (1988) identified the mental models used by expert fliers with protocol analysis, for use in flight simulator training. Soloway (1986) predicted that protocol methodology will become the major source of data in studying initial aspects of programming practices.

Ericsson and Simon (1984) have developed a set of procedures and guidelines for collecting and analyzing protocol analysis data. The use of protocol analyses to infer cognitive processes is based on theoretical assumptions about human cognition. The major assumption is that the same cognitive processes which generate other recordable responses also generate verbalizations (Ericsson & Simon, 1984). This implies that information can be reported only if it is attended to. New information and immediate results of mental operations can be vocalized directly. Information from long-term memory, however, is subject to memory consolidation over time, selective retrieval, and inferential processing. Therefore, concurrent reports (reports given during problem solving) are assumed to be more accurate and complete than retrospective reports (reports given after problem solving) (Ryder, Redding, & Beckschi, 1988).

The first uses of verbal data in research were criticized as being unreliable. Some have argued (Cooke & McDonald, 1986; Nisbett & Wilson,

1977) that subjects are not aware of relevant information during experiments and therefore cannot make accurate reports. These researchers found that people sometimes base answers on inferences about what they think, rather than relying on memory. Ericsson and Simon (1984) point out that these studies would have yielded considerable insight had better probing methods been used. The validity of a verbal report depends greatly upon the questions and/or instructions used to elicit the report. Therefore, protocol analysis requires careful planning, data collection, and analysis to maximize the likelihood of obtaining reliable information (Ryder et al., 1988).

Another criticism of verbal data has been that the cognitive processes themselves could be affected by requiring verbalization. The effect of verbalization on cognitive processes is once again dependent upon the instructions given to the subjects. There are three levels of verbalization (Ericsson & Simon, 1984). The first level is vocalization of a thought that is already verbally encoded. When the contents of short-term memory are words, the words can be spoken without interfering with ongoing cognitive processes. This level takes no special effort and places no additional demands on processing time or capacity. Performance time in nearly all studies reviewed was the same for both verbalization and control conditions. The second level involves description of thoughts. Some information is stored in memory in non-verbal form; visually encoded thoughts, for example. Thought in this form can proceed much faster than speech. Complete verbalization of non-verbal thoughts takes time and thus slows down the thinking itself. No new information is introduced, but processing time is required to label existing information. At the third level, thought processes are not only vocalized, but also explained. Subjects are reporting thoughts about information that was previously attended to and may therefore not be as complete or accurate. Subjects may also incorrectly infer the experimenter's motives or causes (Ericsson & Simon, 1984).

Therefore, it is up to the experimenter to ensure that instructions elicit verbalizations in levels one and two only. The most important distinction is whether the instructions require the subjects to merely describe their thoughts or to explain them. The accuracy of protocol

analysis depends upon not asking for anything that requires interpretation. Reprocessing may change the thought processes themselves.

Warm-up problems can be used to get subjects to think aloud and become comfortable with the microphone (Ericsson & Simon, 1984). During warm-up problems, the experimenter can interrupt and explain whether the subjects are verbalizing correctly or incorrectly. This helps the subjects to understand what is required of them. During the actual experiment, any reminders to keep talking need to be kept short so they will not interfere with the subject's processing.

Experimenters must also recognize that the processes underlying some behavior, like recognition, may be unconscious and therefore not reportable. The results of these processes, however, can still help to clarify strategies used, inferences made, and what is recognized (Ericsson & Simon, 1984).

Ericsson and Simon (1984) have outlined four steps in protocol analysis. First, a tape recording is made of people who are thinking aloud while solving problems. The tape recording is then transcribed into individual statements. Next, the statements are encoded into previously determined theoretical categories. When behavior commonalities need to be determined, the statements can be coded at more aggregated levels than individual statements. Finally, the encoded data are analyzed to determine the reasoning strategies used. Specific comparisons can be made among the protocols. An informal analysis can also be performed on protocol data, without encoding or a priori hypotheses, to obtain information about problem-solving methods (Ryder et al., 1988).

II. METHODOLOGY

Subjects

No small sample of programmers can be considered representative of the population of programmers, since no standard type of individual becomes a programmer. Education, job title, experience, and work skills differ radically (Guidon et al., 1987). In prior debugging studies, each researcher established his own set of criteria for determining who was an experienced programmer. Often college seniors or graduate students in computer science, with little professional experience, were classified as

experts. The present research used two separate standards for defining expert programmers. First, these subjects had to be programmers with at least 2 years of professional programming experience. Second, the subjects had to complete the required tasks in a specified amount of time, which was determined through pre-tests to be insufficient for novices.

Twelve professional C programmers volunteered to serve in this experiment. C was chosen because of its wide usage and the availability of subjects. All of the programmers were connected with the Operations Training Division of the Air Force Human Resources Laboratory and employed by Government contractors. All had at least 2 years of professional programming experience and C was the preferred programming language of each.

Equipment

The subjects were placed in a closed office for the experiment. They did their programming on a video terminal attached to a VAX computer with the VMS operating system. The operating system was augmented to save a copy of the source code every time a subject exited the EDT editor. The subjects were given pencil and paper for scratch work, along with a calculator and a copy of the initial program. For reference, they had editor usage instructions, debugger usage instructions, and two VAX C programming manuals. A printer was also supplied.

Protocol data were collected by recording the subjects' vocalized thoughts. As asserted by Ericsson and Simon (1984), experts sometimes automate portions of their task performance to the point that they lose conscious knowledge of some aspects of the task; thus, the explanations may not match their actual processing mechanisms. Therefore, in addition to recording each subject's verbalizations, the video output from the terminal was captured on videotape. The video output was used to fill in gaps when the subject did not vocalize some of the coding.

Procedures

For a protocol analysis, Ryder et al. (1988) noted that the problems need to be typical or representative of the job domain, taking time and subject availability into account. Ryder et al. also pointed out that for complex tasks, preliminary evaluation is necessary to select tasks that are

sufficiently difficult. The tasks should represent bottlenecks, yet not be so difficult that no one knows where to begin. On that basis, four complex, yet do-able programming tasks were chosen for this experiment.

The first task was to sum an array of numbers and output the results to a file. This task was intended to be a warm-up task, giving the programmers time to become accustomed to the programming environment. The second task was to find the square root of the sum to within .01, without using any built-in square root functions. This task tests the ability to converge on a given value. The third task required that each number in the array be converted to octal. This task tested the ability to use different representations. The final task was to read-in a number from the terminal and use a binary search to locate the number in the sorted array. Although the concept is simple, Knuth (1971) found that over 80% of experienced programmers write the logic for a binary search incorrectly the first time. This task seemed appropriate, because the present research is studying debugging techniques.

Shneiderman (1986) stated that it is necessary for researchers to conduct at least one pilot test of materials and procedures for experiment refinement. Protocols were collected from part-time programmers before the actual experiment began. They were given an unlimited amount of time to complete all four tasks while vocalizing their thoughts. Two of the four programmers completed the tasks in under 6 hours. The other two gave up after 4 hours. Mental and physical fatigue seemed to affect each subject's performance and attitude after the 3 1/2-hour point. All of the subjects spent a great deal of time trying to figure out where to begin on the square root problem, and several had forgotten how the octal-based numbering system worked.

Based on these findings, the instructions were modified somewhat for the actual experiment. The final experimental procedure was as follows. The subjects were first asked to fill out a background questionnaire and answer a few questions about their personal experience with debuggers (see Appendix A). They were then introduced to the experiment setup and asked three warm-up questions (see Appendix B) to give them practice in thinking aloud. When the subjects were comfortable thinking aloud, they were given a set of written instructions (see Appendix B) and asked to complete three

programming tasks. The tasks were to sum an array of numbers, convert each number to octal, and perform a binary search for an input number. The subjects were instructed to think aloud and the experimenter prompted them if they remained quiet for too long. The subject's voice was recorded on the audio track of a videotape. The video output from the subject's terminal was captured on the video track of the same tape.

The time limit was 3 1/2 hours. A one-line time stamp appeared on the subject's screen every 10 minutes. It said, "TIME x:xx.xx - If in Editor, press CTRL-W to refresh screen." Although the time stamp interrupt may have momentarily affected the subject's thought process, this study did not address that issue. Interrupts of this type are not uncommon in programming environments. Timing and taping began when the subject started reading the instructions and ended when he was finished and the experimenter had tested the complete program.

III. RESULTS

Questionnaire Results

Background Variables

Of the 12 subjects who took part in this study, only eight completed all three programming tasks. By the criteria listed in Section II, these eight were classified as experts and their data were further analyzed. The eight who finished and the four who did not were also compared on variables

Table 1. Background Comparison of Programmers Who Finished the Tasks vs. Programmers Who Did Not Finish the Tasks

	Finished (n = 8)			**Did Not Finish** (n = 4)		
Background Variable	Mean	Min	Max	Mean	Min	Max
Yrs work with computers	8.6	2	18	7.8	3	12
Yrs professional programming	4.4	2	10	3.3	2	5
Hrs/wk spent programming	22.5	15	30	20.0	0	30
Number programming languages	5.5	4	9	6.0	4	10
Number operating systems	5.0	3	10	4.3	2	8
Yrs post-secondary education	4.0	2	6	4.3	4	5

measuring background experience (see Table 1). The programmers who were not able to complete the tasks had averaged slightly fewer years working with computers and programming.

Debugger Use

Various debuggers were available for use by all of the subjects in their current jobs. However, three of the programmers stated that they had never used a debugger to help them debug their code. Reasons given were that they did not think the debugger would be very useful since they usually debugged their code fairly quickly without it, and that they did not have time to learn how to use a debugger. One subject simply stated that he made few mistakes.

The other nine programmers had used at least one debugger. Their usage experience included the VAX debugger on VMS, Microsoft Codeview on MS-DOS, sdb on UNIX and DBx Tool on Sun Workstations. Most of the programmers said that they learned the debugger well enough to use it productively within one week. Still, these programmers tended to use the debuggers only as a last resort. They stated that they use debuggers when other methods do not work; e.g., after debug print statements and hand-simulation.

The most frequently used debugger features were displaying variable and register values, setting breakpoints, and single-stepping through code. Programmers complained about the lack of meaningful mnemonics when assigning control keys to commands (e.g., F7 = STEP, rather than CTRL-S).

Programming/Debugging Results

All the subjects attempted the tasks in the given numbered order, completing each task before proceeding to the next task. Although the VAX interactive debugger was available, none of the subjects used it. The four subjects who were stopped at the 3 1/2-hour point were nowhere near completion. None of them had finished task two; so they had not even attempted task three. The other eight subjects completed all three tasks. As each task was finished, the completion time and the number of times the editor was used were noted (see Table 2). A task was considered complete when the working version was saved in the editor. The amount of time spent

on each task varied for tasks one and two, but the time spent on task three was near 1 hour for all eight subjects.

Table 2. Programming/Debugging Times and Number of Editor Uses

SUB	Time to Complete Each Task				Number of Times Editor Used			
	Tsk 1	Task 2	Task 3	Total	Task 1	Task 2	Task 3	Total
#1	10 min	28 min	56 min	1 hr 34 min	1 edit	4 edits	9 edits	14 edits
#2	27 min	32 min	1 hr 3 min	2 hr 2 min	10 edits	6 edits	11 edits	27 edits
#3	6 min	36 min	57 min	1 hr 39 min	1 edit	7 edits	6 edits	14 edits
#4	10 min	38 min	1 hr 2 min	1 hr 50 min	3 edits	9 edits	8 edits	20 edits
#5	9 min	27 min	1 hr 1 min	1 hr 37 min	5 edits	7 edits	8 edits	27 edits
#6	59 min	1 hr 2 min	1 hr 4 min	3 hr 5 min	18 edits	8 edits	15 edits	41 edits
#7	13 min	1 hr 26 min	1 hr 14 min	2 hr 53 min	2 edits	15 edits	6 edits	23 edits
#8	21 min	1 hr 40 min	1 hr 34 min	3 hr 30 min	3 edits	12 edits	10 edits	25 edits
AVG	20 min	50 min	1 hr 6 min	2 hr 16 min	5.4 edits	8.5 edits	10 edits	23.9 edits

Before the protocol data could be analyzed, the possible cognitive activities that could occur during the session had to be enumerated. The problem space and universe of operators can be defined either through pilot work or after transcription of the verbal protocol prior to encoding (Ryder et al., 1988). The coding categories were generated based on the pilot study trials, previous studies, and programming knowledge (see Table 3).

The categories were then formalized using functional notation, as proposed by Ericsson and Simon (1984).

Table 3. Coding Categories

Code	Meaning
CALCULATE	Code calculation of named variable(s)
CHANGE	Change lines of source code as indicated
COMMENT	Verbal comment made by subject
COMPILE	Compile and link named source code
DECLARE	Code declaration of named variable(s)
DELETE	Delete indicated lines of source code
EDIT	Edit named file
ERROR	Named error was produced by compiler
EXIT	Exit from editor and save named file
GOTO	Move to indicated place in source code
HANDCHECK	Hand-simulate code or hand-calculate values
INITIALIZE	Code sets initial value for named variable
INSERT	Insert lines of source code
NEED	Necessary action verbalized by subject
PRINT	Code prints to file or terminal screen
READ	Reading instructions, manual or terminal screen
RUN	Run named executable code
TEST	Run executable code with indicated input
TYPEFILE	Type named file to terminal screen

The protocol data were first transcribed into English phrases and sentences. The phrases and sentences were then coded with functional notations using the categories in Table 3. Any statement that could not be encoded was placed in a catchall category, COMMENT, as suggested by Ryder et al. (1988). Video and source code data were incorporated into the coded data for clarity in the analysis (see Appendix C). The types of errors made and the techniques used to find the errors were then analyzed.

The subjects' coded data were examined to find the errors that had to be debugged. This search began following the first exit from the editor for each task. Any changes made before the first exit were not counted, because a debugger cannot be used until the first version of the source code is typed in. Errors were isolated by working backwards from the changes made to the code. Each error was identified and grouped into one of the classes described in Section I (see Table 4). The syntax errors were subdivided

into those caught by the compiler and those caught by the programmer. The technique the subject used to find the error was also noted.

Table 4. Error Classification

	Subject								
Type of Error	#1	#2	#3	#4	#5	#6	#7	#8	Totals
Appreciation	1	0	0	1	0	1	1	1	5
Lexical/Syntactic									
Found by Compiler	6	5	4	3	3	16	5	7	49
Found by Programmer	3	0	1	0	0	6	1	1	12
Execution	0	2	0	1	0	2	3	0	8
Intent/Logic	4	2	10	10	16	8	6	12	68
Total Errors Made	14	9	15	15	19	33	16	21	142

The five appreciation errors were of two types. Subjects 1, 6 and 7 did not initially format their sum output correctly. Subjects 4 and 8 tried to use the "%o" print format to output the octal representations instead of actually converting the numbers. As expected, most of the lexical/syntactic errors were caught by the compiler. Even the lexical/syntactic errors that the programmers had to find themselves, however, were discovered fairly quickly by simply reading through the code. The most common syntax errors were incorrect spellings, missing punctuation, incorrect use of operators, and missing declarations. The execution and logic errors were the hardest to locate and fix. All of the execution errors occurred due to access violations, when the programmer tried to access an address in memory that was beyond the legal bounds of the program. The most common logic errors were:

1. variables not initialized
2. incorrect array indices
3. variable values calculated incorrectly
4. loop exit condition incorrect
5. digits of the octal representation reversed

6. no code for case when number was not in the array

7. boundary conditions not checked by algorithm

Most of the subjects used the same debugging strategy. After the first version of each task was coded in the editor, the compiler was used to find the syntax errors. Once the syntax errors had been fixed, the programmers ran the program and examined the output. If the output did not look correct, the code was reviewed. If the error was not obvious, debugging print statements were inserted in the code and/or a hand-simulation of the code was carried out on paper. The programmers then ran the program and examined the output again. Eventually, some combination of debug print statements and hand-simulation would reveal the error's cause. After the error was fixed, the program was run again to see the effects of the changes. Overall, the programmers used the code, the output, debug print statements, and hand-simulation to find their bugs. It is interesting to note that many of the bugs were corrected before the first code was even entered. The programmers often hand-simulated their code to a certain extent before coding, revealed their initial errors, and fixed them before they actually typed in the code.

IV. DISCUSSION

This experiment differs from previous debugging studies in that it examines programmers as they debug their own code instead of finding specific bugs placed in an experimental program. Thus it allows for a greater variety of error types, and therefore a more complete study of debugging techniques.

The results of this experiment provide some insight into programmers' use of and attitude toward interactive debuggers. The programmers' debugging techniques, insertion of debug print statements and hand-simulation are supported by most debuggers. Instead of inserting debug print statements, a programmer can use a debugger to show variable values at any point during execution. Similarly, the programmer can use the debugger to step through the code, instead of performing a hand-simulation on paper. The results would be more accurate, in that the system would not overlook any line of code.

Even though using the debugger would have been simpler, none of the subjects in this experiment did so. Half of the subjects were not familiar with the VMS environment, which would explain their reluctance to try a new debugger. The other half did not feel that the program was complicated enough to warrant its use. The debugger was seen only as a last resort, if they could not debug the program by hand.

The results of this experiment imply that debuggers would be used more frequently if they were easier to learn. Therefore, the designers should now concentrate on the presentation of the debugger, as the programmers' functional needs are being met. This area includes interface design, documentation, and marketing. When a piece of software is designed for a naive user, a great deal of time and effort are spent on making the product easy to use. When the market for a software product is computer professionals (proficient computer users), however, manufacturers place less emphasis on ease of use. On the questionnaires though, several subjects expressed interest in simple tutorials for debuggers, as well as more examples to follow.

Product support for software debuggers typically consists of a single reference manual. The manual is usually very thick and covers every command the debugger supports. The results of the present experiment show there are only about five debugger commands that would be used by all programmers. These commands allow the programmer to single-step through a program, set breakpoints, display variables and registers, and set new values. These commands should be made so easy to use that they take only minutes to learn. The debugger documentation should introduce these commands first, instead of burying them in with the rest of the commands. The programmers could then use the debugger productively with very little effort. The more complex, less-used commands could then be learned as they become necessary.

The results of this experiment also indicate that programmers would be interested in specifying certain variables for the debugger to display constantly. The interface could be designed such that a portion of the screen would be reserved for displaying variable values. The programmer would not have to ask to see the value of each variable every time the

execution paused. Before executing the code in the debugger, the programmer would name the variables to be displayed at every stopping point.

There are several ways these features could be implemented that would make them easy to use. Debuggers designed around a command language could ensure that the five most-used commands were accessible using one-letter or one-key commands. Debuggers designed around menu interfaces could include pulldown menus for the most-used features. No matter what method of implementation is employed, ease of use should be the primary concern.

V. CONCLUSIONS AND RECOMMENDATIONS

This research provided some understanding of the debugging techniques used by professional programmers. One of the limitations of studying programmers debugging their own code, however, was that the task complexity had to be limited. The tasks had to be modified so that the programmers could complete them in one sitting. It would be interesting to see if hand-simulation and debug print statements were used as frequently in coding more complex problems. This research did not address integration or portability issues either. The methods used to debug code during an integration effort may be entirely different.

The data for this experiment were collected using only C programmers, a subset of the programming world. Further research should be conducted to see if the results apply to programmers as a whole. The results of this experiment show that debuggers are difficult to learn but are functionally well designed. Future research should focus on the best ways to improve the documentation and user interface, so that they will be used more frequently.

REFERENCES

Beander, B. (1983). VAX DEBUG: An interactive, symbolic, multilingual debugger. SIGPLAN Notices 18, 8, 173-179.

Blackman, H. S. (1988, October). Overview: The use of think-aloud verbal protocols for the identification of mental models. Proceedings of the Human Factors Society - 32nd Annual Meeting (pp. 872-874). Santa Monica, CA.

Brown, A. R., & Sampson, W. A. (1973). Program debugging: The prevention and cure of program errors. New York: American Elsevier.

Carver, S. M., & Klahr, D. (1986). Assessing children's logo debugging skills with a formal model. Journal of Educational Computing Research, 2, 487-525.

Carver, S. M., & Risinger, S. (1987). Improving children's debugging skills. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), Empirical studies of programmers: Second workshop (pp.147-171). Norwood, NJ: Ablex Publishing Corporation.

Cooke, N. M., & McDonald, J. E. (1986). A formal methodology for acquiring and representing expert knowledge. Proceedings of the IEEE, 74, 1422-1430.

Curtis, B. (1986). By the way, did anyone study any real programmers? In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers (pp. 256-262). Norwood, NJ: Ablex Publishing Corporation.

Ericsson, K. A., & Simon, H. A. (1984). Protocol analysis: Verbal reports as data. Cambridge, MA: The MIT Press.

Fisher, C. (1987). Advancing the study of programming with computer-aided protocol analysis. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), Empirical studies of programmers: Second workshop (pp.198-216). Norwood, NJ: Ablex Publishing Corporation.

Gould, J. D. (1975). Some psychological evidence on how people debug computer programs. International Journal of Man-Machine Studies, 7(2), 151-181.

Gould, J. D., & Drongowski, P. (1974). An exploratory study of computer program debugging. Human Factors, 16, 258-277.

Gugerty, L., & Olson, G. M. (1986). Comprehension differences in debugging by skilled and novice programmers. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers (pp. 13-27). Norwood, NJ: Ablex Publishing Corporation.

Guidon, R., Krasner, H., & Curtis, B. (1987). Breakdowns and processes during the early activities of software design by professionals. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), Empirical studies of programmers: Second workshop (pp. 65-82). Norwood, NJ: Ablex Publishing Corporation.

Harris, T. (1988). Software metrics. Proceedings of the Fall 1988 DECUS U.S. Symposium (pp. 17-21). Anaheim, CA.

Holt, R. W., Boehm-Davis, D. A., & Schultz, A. C. (1987). Mental representations of programs for student and professional programmers. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), Empirical studies of programmers: Second workshop (pp. 33-46). Norwood, NJ: Ablex Publishing Corporation.

Jeffries, R. (1982). A comparison of the debugging behavior of expert and novice programmers. Proceedings of the American Educational Research Association.

Kessler, C. M., & Anderson, J. R. (1986). A model of novice debugging in LISP. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers (pp. 198-212). Norwood, NJ: Ablex Publishing Corporation.

Knoke, R. (1988). Debugging embedded C. Embedded systems programming, 1(1), 28-36.

Knuth, D. E. (1971). An empirical study of FORTRAN programs (IBM Research Report No. RC-3276).

Lukey, F. J. (1980). Understanding and debugging programs. International Journal of Man-Machine Studies, 12(2), 189-198.

Nanja, M., & Cook, C. R. (1987). An analysis of the on-line debugging process. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), Empirical studies of programmers: Second workshop (pp. 172-184). Norwood, NJ: Ablex Publishing Corporation.

Nisbett, R. E., & Wilson, T. D. (1977). Telling more than we can know: Verbal reports on mental processes. Psychological Review, 84, 231-259.

Rasmussen, J. (1981). Models of mental strategies in process plant diagnosis. In J. Rasmussen & W. Rouse (Eds.), Human detection and diagnosis of system failures. New York, NY: Plenum Press.

Rasmussen, J. (1986). Information processing and human-machine interaction: An approach to cognitive engineering. New York, NY: North-Holland.

Ryder, J. M., Redding, R. E., & Beckschi, P. F. (1988). Procedural guide for integrating cognitive methods into ISD task analysis (Draft). Horsham, PA: Pacer Systems.

Seviora, R. E. (1987). Knowledge-based program debugging systems. IEEE Software, 4(3), 20-32.

Sheil, B. A. (1981). The psychological study of programming. Computing Surveys, 13(1), 112-116.

Shneiderman, B. (1986). Empirical studies of programmers: The territory, paths, and destinations. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers (pp. 1-12). Norwood, NJ: Ablex Publishing Corporation.

Soloway, E. (1986). What to do next: Meeting the challenge of programming-in-the-large. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers (pp. 263-268). Norwood, NJ: Ablex Publishing Corporation.

Spohrer, J. G., & Soloway, E. (1986). Analyzing the high frequency bugs in novice programs. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers (pp. 230-251). Norwood, NJ: Ablex Publishing Corporation.

Vessey, I. (1985). Expertise in debugging a computer program: A protocol analysis. International Journal of Man-Machine Studies, 23, 459-494.

Weiser, M. (1982). Programmers use slices when debugging. Communications of the ACM, 25, 446-452.

Zelkowitz, M. V., Yeh, R. T., Hamelt, R. G., Gannon, J. D., & Basili, V. R. (1984). Software engineering practices in the US and Japan. Computer, 17(6), 57-65.

APPENDIX A: SAMPLE QUESTIONNAIRE

SAMPLE QUESTIONNAIRE

1. Are you male or female?
2. How old are you?

3. What education level have you achieved?
   (High School, Some College, Associate Degree, Bachelors Degree,
   Some Graduate Courses, Masters Degree, Doctoral Degree...)

4. If you completed a degree or degrees, what was your major field of study? (undergraduate, and graduate if applicable)

5. Did you learn to program on a batch or an interactive system?

6. How many months/years have you been in the computer field?
7. How many months/years have you been **programming professionally?**
8. How many hours per week do you **currently** spend programming?

9. How many different operating systems have you used?
10. Are you familiar with the VAX/VMS operating system?

11. How many programming languages have you programmed in?
12. What is your preferred programming language?

13. Do you ever use a debugger to help debug code?

14. If the answer to #13 was "NO":

    a. Is a debugger currently available for your use?
    b. If "YES," why don't you use it?

15. If the answer to #13 was "YES":

    a. Which debugger(s) have you used?

    b. How many weeks/months did it take you to learn this debugger well enough to use it productively?

    c. Under what circumstances do you use the debugger?

    d. What debugger features do you use most frequently?

    e. What features in this debugger (or these debuggers) are the most cumbersome?

    f. How would you suggest that they be improved?

    g. What additional features would you like to see incorporated in debuggers?

16. What methods, other than interactive debuggers, do you use to debug code?

APPENDIX B: STUDY INSTRUCTIONS

VERBAL SCRIPT

Please fill out the questionnaire.

Warm-Up Problems:

In this experiment we are interested in what you think about when you program, test, and debug. We are especially interested in what you are thinking whenever you notice anything wrong - either in your logic or your coding, and how you find and correct the error. Therefore, I want you to THINK ALOUD as you work on each task.

What I mean by think aloud is that I want you to tell me everything that passes through your head from the time you first read the instructions for the task until you have a working version of the program. TALK CONSTANTLY. I don't want you to plan out what you say or try to explain to me what you are saying. Just act as if you are alone in the room speaking to yourself.

IT IS VERY IMPORTANT THAT YOU KEEP TALKING. If you are silent for a long period of time, I will remind you to keep talking. ALSO SPEAK LOUDLY. Do you understand what I want you to do?

To get you used to the idea of thinking aloud, I'm going to ask you a few warm-up questions. I want you to think aloud as you figure out the answers.

1. **How much is 384 divided by 16? (24)**

Good. Now I want to see how much you can REMEMBER about what you were thinking from the time you heard the question until you gave the answer. If possible I would like you to tell me about your memories in the sequence they occurred while working on the question. I don't want you to work on solving the problem again; just report all that you can remember thinking about when answering the question.

Good. I will give you two more practice problems. I want you to do the same thing for each of these problems. Here's your next problem.

2. **How many windows are there in your house?**

Now tell me all that you can remember about your thinking.

Here is your last practice problem. Think aloud as you answer. There is no need to keep count. I will keep track for you.

3. **Name 20 animals.**

Now tell me all that you can remember about your thinking.

VERBAL SCRIPT

Main Experiment:

You will be given three programming tasks to do. Complete the tasks in any order. If you have a trouble solving any problem, proceed to the next task, and come back to it later.

Do you have:

1. Hardcopy of the source code?
2. File containing source code?
3. Copy of the desired output?
4. Experiment Instructions?
5. Paper and Pencil?
6. Calculator?
7. Terminal and Printer Access?
8. "Use of Compiler & Editor" Instructions?
9. Editor Keys Handout?
10. Debugger Instructions?
11. Language Manuals?

Warning:

Your screen will be time-stamped every 10 minutes.
If you are in the editor, use Control-W to refresh your screen.
Ignore the time stamp and continue to work.

Final Reminders:

If you are typing in or reading your code, you can read aloud.

TALK CONSTANTLY
SPEAK LOUDLY

WRITTEN INSTRUCTIONS

Programming Tasks:

Expand the functionality of the program in file PROG.C so that it will do the following (NOTE: Do NOT create a new file. Edit the existing PROG.C file):

1. Compute the **sum** of all of the numbers in the array. Output the sum to a results file. See the attached example of desired output. **(REMEMBER TO THINK ALOUD)**

2. **Convert** each number in the array **to octal.** See the conversion examples below. (Note: You must actually convert each number, not just output the number with the %o.) Output the decimal and octal conversion representations of each entry to the results file. See the attached example of desired output. **(REMEMBER TO THINK ALOUD)**

   In decimal, digits signify the quantity of each power of 10.

   i.e. $937_{10} = (9 \times 10^2) + (3 \times 10^1) + (7 \times 10^0)$

   In octal, the digits work the same way.

   i.e. $937_{10} = 1651_8 = (1 \times 8^3) + (6 \times 8^2) + (5 \times 8^1) + (1 \times 8^0)$

   Conversion By Division:

   937 / 8 = 117 rem 1
   117 / 8 = 14 rem 5
   14 / 8 = 1 rem 6
   1 / 8 = 0 rem 1

   Conversion By Subtraction:

   $937 - (1 \times 8^3) = 425$
   $425 - (6 \times 8^2) = 41$
   $41 - (5 \times 8^1) = 1$
   $1 - (1 \times 8^0) = 0$

3. Prompt for a number between 1 and 50 from the terminal. Use a **binary search** to locate the number in the given sorted array.

   Binary search:

   -Compare input number to middle entry of array.
   -If input number is smaller, repeat search on first half of array.
   -If input number is larger, repeat search on second half of array.
   -Repeat search until input number is found or determined not to exist in array.

   If the input number is found, output the number and the array index where it is located to the terminal. If the input number is not found, output a message stating that fact. **(REMEMBER TO THINK ALOUD)**

WRITTEN INSTRUCTIONS

Initial Program:

```
#include <stdio.h>

main()
/*  This program initializes array of integers.  */
{
   int nums[20] = {1,3,5,8,9,12,13,19,25,26,27,29,33,35,38,39,40,43,45,50};
   printf ("The NUMS array has been initialized.\n");
}
```

Desired Output:

In the RESULTS file (for tasks 1 and 2):

Array Sum = n

Array Index	Decimal Rep	Octal Rep
n	n	n
.	.	.
.	.	.
.	.	.
n	n	n

To the TERMINAL (for task 3):

Input number, n, found at array index n.

or

Input number, n, does not exist in the array.

WRITTEN INSTRUCTIONS

Compiler and Editor:

EDITOR: To edit the file:

$ EDIT PROG.C

To exit the editor and save your file:

Hit the <F10> key.

COMPILER: *To compile, link and run the file:*

```
$ CC PROG        (with listing file:  $ CC/LIST PROG)
$ LINK PROG, SYS$LIBRARY:VAXCRTL/LIBRARY
$ RUN PROG
```

To compile, list, and link at the same time:

$ CLL PROG

To compile, link and run the file with the debugger:

```
$ CC/DEBUG PROG
     (with listing file:  $ CC/DEBUG/LIST PROG)
$ LINK/DEBUG PROG
$ RUN PROG
```

To compile, list, and link with the debugger:

```
$ CLD PROG
```

Miscellaneous VAX Commands:

DIRECTORY LISTING: To display names of files in your directory:

```
$ DIR
```

FILE CONTENTS: To display the contents of a file in your directory:

```
$ TYPE filename
```

HOLD SCREEN: To pause the scrolling of output on your screen, press:

<F1> key (Press again to resume output display.)

APPENDIX C: SAMPLE SUBJECT DATA

DATA SOURCES:

P = Protocol (Verbal) Data
S = Source Code (from Edit Sessions) Data
V = Videotape Data

CODING CATEGORIES:

CALCULATE	- Code calculation of named variable(s)
CHANGE	- Change lines of source code as indicated
COMMENT	- Verbal comment made by subject
COMPILE	- Compile and link named source code
DECLARE	- Code declaration of named variable(s)
DELETE	- Delete indicated lines of source code
EDIT	- Edit named file
ERROR	- Named error was produced by compiler
EXIT	- Exit from editor and save named file
GOTO	- Move to indicated place in source code
HANDCHECK	- Hand-simulate code or hand-calculate values
INITIALIZE	- Code sets initial value for named variable
INSERT	- Insert lines of source code
NEED	- Necessary action verbalized by subject
PRINT	- Code prints to file or terminal screen
READ	- Reading instructions, manual or terminal screen
RUN	- Run named executable code
TEST	- Run executable code with indicated input
TYPEFILE	- Type named file to terminal screen

SUBJECT #2 DATA

TIME 0:00:00 - TASK #1

```
P     READ (INSTRUCTIONS FOR TASK 1)
PV    EDIT (PROG.C)

PV    INSERT (COMMENT)
S        /* sum the array elements */

P     COMMENT (THERE'S 20 OF THEM, SO WE'LL MAKE A LOOP OF 20)

PV    INSERT (FOR LOOP)
S        for (i=0; i=20; i++)

PV    DECLARE (INTEGER, I)
S        int i;

P     NEED (PLACE TO COLLECT THE SUM)
PV    DECLARE (INTEGER, SUM)
PV    INITIALIZE (SUM)
S        int i, sum=0;

PV    CALCULATE (SUM, FOR EACH ELEMENT)
S        sum += num[i];
P     COMMENT (ONLY ONE LINE IN THERE, SO WE DON'T NEED BRACKETS)

PV    PRINT (MESSAGE TO MAKE SURE WE GOT THROUGH THERE) TO SCREEN
S        printf ("The sum of the NUMS array as been computed");
PV    EXIT (PROG.C)

PV    COMPILE (PROG.C)
PV    ERROR (NUM IS NOT DECLARED WITHIN THE SCOPE OF THIS USAGE)
P     COMMENT (IT'S NUMS, NOT NUM)

PV    EDIT (PROG.C)
PV    CHANGE (NUM TO NUMS IN SUM CALCULATION)
S        sum += nums[i];
PV    EXIT (PROG.C)

PV    COMPILE (PROG.C)
PV    RUN (PROG)
P     READ (SCREEN - NUMS ARRAY HAS BEEN INITIALIZED... NOTHING ELSE?)

PV    EDIT (PROG.C)

P     COMMENT (OH, MY < DIDN'T MAKE IT)
PV    CHANGE (INSERT < IN THE FOR LOOP EXIT CONDITION)
S        for (i=0; i<=20; i++)
PV    EXIT (PROG.C)

PV    COMPILE (PROG.C)
P     READ (SCREEN - ...HAS BEEN INITIALIZED AND COMPUTED.  GOOD)
P     COMMENT (GET BACK IN THE EDITOR AND WRITE IT TO A FILE)

PV    EDIT (PROG.C)
P     NEED (TO WRITE THE RESULT TO A FILE)
PV    INSERT (COMMENT)
S        /* write the result to a file */

P     COMMENT (LET'S PRINT THE RESULTS TO THE SCREEN ALSO)
PV    PRINT (SUM) TO SCREEN
S        printf("The sum of the NUMS array is %d\n", sum);
```

TIME 0:10:00 - TASK #1

```
PV    INSERT (OPEN FILE, RESULT.DAT, RETURNING A FILE POINTER)
P     READ (INSTRUCTIONS FOR TASK 1 ON NAME OF OUTPUT FILE)
PV    CHANGE (OPEN IN WRITE MODE)
S        fp = fopen("result.dat", "w+");

P     READ (MANUAL - LOOKING FOR FOPEN ERROR CHECKING)
P     COMMENT (CAN'T FIND IT, SO I'LL ASSUME THE OPEN WORKED)

PV    PRINT (SUM) TO FILE
S        fprintf(fp, "The sum of the NUMS array is %d\n", sum);

PV    INSERT (CLOSE FILE)
```

```
S       fclose(fp);
PV   EXIT (PROG.C)

PV   COMPILE (PROG.C)
PV   ERROR (FP NOT DECLARED WITHIN THE SCOPE OF THIS USAGE)

PV   EDIT PROG.C
PV   DECLARE (FILE POINTER, FP)
S       FILE *fp;
PV   EXIT (PROG.C)

PV   COMPILE (PROG.C)
PV   RUN (PROG.C)

P    READ (SCREEN - SUM IS 1012)
P    HANDCHECK (ADD ON CALCULATOR - 500?)

P    COMMENT (LET'S SEE WHAT MY DATA FILE LOOKS LIKE)
PV   TYPEFILE (RESULT.DAT)
P    COMMENT (SAME THING - MAYBE SUM WASN'T GETTING INITIALIZED)
P    COMMENT (LET'S SEE WHAT SUM WAS BEFORE I STARTED ADDING)

PV   EDIT (PROG.C)
PV   GOTO (BEFORE COMPUTATION OF SUM)
PV   PRINT (SUM) TO SCREEN
S       print("Before computation, sum was %d\n",sum);
PV   EXIT (PROG.C)

PV   COMPILE (PROG.C)
PV   ERROR (UNDEFINED SYMBOL, PRINT)
P    COMMENT (I BET I TYPED THAT WRONG - PRINTF WOULD HELP)

PV   EDIT (PROG.C)
PV   CHANGE (PRINT TO PRINTF)
S       printf("Before computation, sum was %d\n",sum);
PV   EXIT (PROG.C)

PV   COMPILE (PROG.C)

TIME 0:20:00 - TASK #1

PV   RUN (PROG)
P    COMMENT (STILL GETTING 1012.  COULD I HAVE ADDED WRONG?)
P    HANDCHECK (ADD ON CALCULATOR... 500)

PV   EDIT (PROG.C)
P    COMMENT (I GUESS I CAN LOOK AT SUM EACH TIME)
PV   PRINT (SUM, I, AND NUMS[I]) TO SCREEN EACH TIME
S       printf("i=%d, sum=%d, nums[i]=%d\n",i, sum, nums[i]);
PV   EXIT (PROG.C)

PV   COMPILE (PROG.C)
PV   RUN (PROG)
P    COMMENT (I'M STOPPING AT 20 - SHOULD STOP AT 19)
P    COMMENT (BUT THE SUM WORKED. WHY DID THE PRINT STATEMENT FIX IT?)

**EXPERIMENTOR NOTE:  THE VALUE IN NUMS[20] FOR THIS SUBJECT IS UNRELIABLE SINCE IT IS
                      NOT PART OF THE DECLARED ARRAY.  ON THIS RUN, IT HAPPENED TO BE
                      EQUAL TO ZERO, AND THEREFORE DID NOT HURT THE FINAL SUM.

PV   EDIT (PROG.C)
PV   CHANGE (FOR LOOP EXIT CONDITION FROM <20 TO <19)
S       for (i=0; i<20; i++)

P    COMMENT (BUT PRINTING SOMETHING, SHOULDN'T CHANGE THE ANSWER...)
PV   CHANGE (COMMENT OUT THE PRINT)
S       /*printf("i=%d, sum=%d, nums[i]=%d\n",i, sum, nums[i]);*/
PV   EXIT (PROG.C)

PV   COMPILE (PROG.C)
PV   RUN (PROG)
P    COMMENT (500.  LOOKS GOOD.  GO CLEAN OUT THE JUNK)

PV   EDIT (PROG.C)
PV   DELETE (TWO DEBUG PRINT STMTS)
P    COMMENT (RUN IT ONE MORE TIME)
PV   EXIT (PROG.C)

PV   COMPILE (PROG.C)
P    READ (INSTRUCTIONS FOR TASK 2)
```

```
PV    RUN (PROG)
P     COMMENT (OK, ONE WORKS.  2ND PROGRAM...)

TASK 2

PV    EDIT (PROG.C)
P     NEED (RESULTS IN THE SAME FILE)
PV    CHANGE (MOVE THE CLOSE DOWN, SO IT'S AT THE END OF THE PROGRAM)

PV    INSERT (TWO COMMENTS)
S        /* 1st project */
S        /* 2nd project */

P     NEED (TO GO THROUGH EACH NUMBER AND CONVERT IT)
PV    INSERT (FOR LOOP)
S        for (i=0; i<20; i++) {
S             }

TIME 0:30:00 - TASK 2

P     COMMENT (DO A FUNCTION TO CONVERT)
PV    CHANGE (FORMAT OF SUM OUTPUT AND ADD A NEWLINE)
S        fprintf(fp, "Array Sum = %d\n\n", sum);

P     NEED (TO PRINT INDEX ARRAY, DECIMAL, AND OCTAL)
P     NEED (A HEADER BEFORE THIS)

PV    PRINT (ARRAY, DECIMAL, OCTAL, INDEX, REP, REP, NEW LINE) TO FILE
PV    INSERT (COMMENT)
S        /* print header to file */
S        fprintf(fp, "Array \t Decimal \t Octal\n",
S                    "Index \t Rep     \t Rep\n");

PV    PRINT (I AND NUMS[I]) TO SCREEN
P     NEED (A FUNCTION WHICH RETURNS CHARACTER POINTER)
PV    CHANGE (ADD CONVERT TO OCTAL FUNCTION TO PRINT STATEMENT)
S        printf("%d \t %d \t %s\n", i, nums[i], conv_to_octal);

PV    CHANGE (PRINT TO FILE INSTEAD OF TO SCREEN)
S        fprintf(fp,"%d \t %d \t %s\n", i, nums[i], conv_to_octal);

P     COMMENT (WHOOPS, I NEED TO GIVE THE FUNCTION AN ARGUMENT)
PV    CHANGE (ADD ARGUMENT NUMS[I])
S        fprintf(fp,"%d \t %d \t %s\n", i, nums[i], conv_to_octal(nums[i]));

P     INSERT (COMMENT ABOUT FUNCTION)
S        /* convert integer argument to octal string */

PV    DECLARE (INTEGER, CONV_TO_OCTAL WITH PARAMETER ARG)
PV    INSERT (THE FUNCTION OUTLINE)
S        int conv_to_octal(arg)
S            int arg;
S            {
S            } /* end of conv_to_octal */

PV    INSERT (COMMENT TO MARK THE END OF MAIN)
S        } /* end of main */

P     NEED (TO TELL IT WHAT THE FUNCTION RETURNS)
P     COMMENT (IF RETURN PTR TO CHAR, HAVE TO GET SPACE FROM SOMEPLACE)
P     COMMENT (MAKE IT AN INTEGER)

PV    GOTO (OUTPUT STATEMENT IN MAIN)
PV    CHANGE (%S TO %D)
S      fprintf(fp,"%d \t %d \t %d\n", i,nums[i],conv_to_octal(nums[i]));

PV    GOTO (CONV_TO_OCTAL FUNCTION DECLARATION)
P     COMMENT (MAKE IT A DUMMY SUBROUTINE FOR NOW)
PV    INSERT (RETURN ITS ARGUMENT)
S            return(arg);

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    RUN (PROG)

TIME 0:40:00 - TASK #2

PV    TYPEFILE (RESULT.DAT)

P     NEED (TO TAKE OUT SPACES SO OCTAL COLUMN WILL LINE UP)
```

```
P     NEED (TO PUT IN AN OCTAL FORMAT COLUMN JUST TO CHECK MY ANSWERS)

PV    EDIT (PROG.C)
PV    CHANGE (DELETE SPACE BEFORE TAB)
S        fprintf(fp, "Array \t Decimal\t Octal\n",
S                    "Index \t Rep     \t Rep\n");

PV    CHANGE (ADD A COLUMN TO PRINT NUMS[I] IN OCTAL FORMAT)
S        fprintf(fp,"%d \t %d \t %d \t %o\n",
S                i, nums[i], conv_to_octal(nums[i]),nums[i]);

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    RUN (PROG)

PV    TYPE (FILE RESULT.DAT)
P     COMMENT (IT DID OCTAL, BUT MY HEADING STILL DIDN'T GET MOVED OVER)

PV    EDIT (PROG.C)
P     COMMENT (I GUESS IT DOESN'T LIKE DOING THE JOINED LINES COMMAND)
PV    CHANGE (COMBINE THE LINES)
S        fprintf(fp, "Array \tDecimal\t Octal\nIndex \tRep     \t Rep\n");
PV    EXIT (PROG.C)

PV    COMPILE (PROG.C)
PV    RUN (PROG)

PV    TYPEFILE (RESULT.DAT)
P     COMMENT (HEADERS LOOK LINED UP NOW)
P     NEED (TO WRITE THE FUNCTION)

P     COMMENT (HOW DOES THE OCTAL COMPUTATION WORK?)
P     COMMENT (DIVIDE, GET A REMAINDER, AND DO IT IN REVERSE ORDER)
P     COMMENT (OR SUBTRACT AND GET IT IN LEFT TO RIGHT ORDER,
                BUT I HAVE TO KNOW WHAT POWER OF 8 TO START ON)
P     COMMENT (DO THE DIVISION METHOD)

PV    EDIT (PROG.C)
P     COMMENT (START WITH MY ARGUMENT)
PV    CHANGE (COMMENT, FROM OCTAL STRING TO OCTAL INTEGER)
S        /* convert integer argument to octal integer */

P     NEED (TO KEEP THE NUMBER THAT WAS DIVIDED BY 8)
P     COMMENT (KEEP MY RESULTS SOMEPLACE, SO LET'S CALL THAT RESULT)

PV    DECLARE (INTEGER, RESULT)
S        int result;

P     NEED (TO KNOW WHAT POWER OF 10 PLACE THIS IS GOING TO GO IN)
PV    DECLARE (INTEGER, POWER_OF_TEN)
S        int power_of_10;

PV    INITIALIZE (RESULT AND POWER_OF_TEN)
S        int result = 0;
S        int power_of_10 = 1;

P     NEED (TO KEEP DIVIDING AS LONG AS ARG > 0)
PV    INSERT (DO WHILE LOOP)
S        do {
S           } while (arg > 0);

TIME 0:50:00 - TASK #2

P     NEED (TO GET THE REMAINDER OF ARG/8, AS WELL AS THE RESULT)
PV    COMMENT (THE REMAINDER I CAN GET WITH THE MOD)

PV    CALCULATE (RESULT, ARG, AND POWER_OF_TEN)
S           result += power_of_10 * (arg % 8);
S           arg = arg / 8;
S           power_of_10 *= 10;

P     HANDCHECK (SO MY ARGUMENT IS 937... IF I MOD THAT BY 8 I GET 1)
P     HANDCHECK (1 TIMES 1, RESULT IS 1.  THEN MODIFY ARG.  DIVIDE BY 8)
P     HANDCHECK (CHANGE POWER_OF_TEN FOR NEXT TIME, =* 10, EQUALS 10)
P     HANDCHECK (ARG IS NOW 117.  DIVIDE... MOD OF 117 MOD 8 IS 5)
P     HANDCHECK (SO ADD 10*5, WHICH IS 50, TO GET THE SECOND DIGIT)
P     HANDCHECK (LAST TIME... START WITH 14, I GET 6.  SO I ADD 600)
P     HANDCHECK (DIVIDE 14 BY 8 GET 1, POWER_OF_TEN BY 10 AND GET 1000)
P     HANDCHECK (DO MOD AGAIN FOR REMAINDER OF 1, THE LAST DIGIT)
```

```
PV    CHANGE (DON'T WANT TO RETURN ARG... WANT TO RETURN RESULT)
S        return (result);

P     COMMENT (VERIFY ALL VARIABLES ARE DEFINED AND INITIALIZED)
PV    EXIT (PROG.C)

PV    COMPILE (PROG.C)
PV    RUN (PROG)
PV    TYPEFILE (RESULT.DAT)
P     COMMENT (LOOKS RIGHT.  GET RID OF THE EXTRA COLUMN)

PV    EDIT (PROG.C)
PV    CHANGE (MOVE OUTPUT OF SUM TO BOTTOM OF TASK 1)

PV    PRINT (MESSAGE INDICATING CONVERSION IS DONE) TO SCREEN
S        printf("Octal conversion complete.\n");

P     COMMENT (LEAVE %O COLUMN IN PRINT TO MAKE SURE I DIDN'T HURT IT)
PV    EXIT (PROG.C)

PV    COMPILE (PROG.C)
PV    RUN (PROG)
PV    TYPEFILE (RESULT.DAT)
P     COMMENT (OCTAL COLUMN STILL MATCHES %O COLUMN.  LOOKS CORRECT)
P     NEED (TO GET RID OF THAT EXTRA COLUMN IN THE OUTPUT NOW)

PV    EDIT (PROG.C)
PV    CHANGE (REMOVE EXTRA COLUMN FROM FILE OUTPUT STATEMENT)
S        fprintf(fp,"%d \t %d \t %d\n",
S                i, nums[i], conv_to_octal(nums[i]));
PV    EXIT (PROG.C)

PV    COMPILE (PROG.C)
P     READ (INSTRUCTIONS FOR TASK 3)
PV    RUN (PROG)
PV    TYPEFILE (RESULT.DAT)
P     COMMENT (ARRAY SUM AND THREE COLUMNS. 9 COMES OUT AS 11. GOOD)

TIME 1:00:00 - TASK #3

PV    EDIT (PROG.C)
P     READ (INSTRUCTIONS FOR TASK 3)

PV    INSERT (FOUR COMMENTS TO AN OUTLINE TASK 3)
S        /* 3rd project */
S        /* get number from terminal */
S        /* make sure number between 1 and 50 */
S        /* find the number in the array */

P     NEED (TO GET THE NUMBER FROM THE TERMINAL)
PV    PRINT (PROMPT WITHOUT A CARRIAGE RETURN) TO SCREEN
PV    INSERT (SCANF FOR AN INTEGER AND PUT IT IN NUMBER)
S        printf("Input a number between 1 and 50: ");
S        scanf("%d", &number);

PV    GOTO (TOP DECLARATIONS)
PV    DECLARE (INTEGER, NUMBER)
S        int number;

PV    GOTO (AFTER SCANF CODE)
PV    PRINT (ECHO INPUT NUMBER BACK) TO SCREEN)
S        printf("number = %d\n", number);

P     COMMENT (MAKE SURE THE NUMBER IS A GOOD ONE)
PV    INSERT (IF STATEMENT TO CHECK INPUT NUMBER)
S        if (number < 1 || number > 50)

P     NEED (TO DO THIS UNTIL THEY ENTER A NUMBER THAT'S RIGHT)
PV    INSERT (A DO WHILE LOOP)
S        do {
S             (PREVIOUS CODE)
S           } while (number < 1 || number > 50);

P     NEED (TO PRINT A MESSAGE BEFORE ASKING THE QUESTION AGAIN)
PV    PRINT (MESSAGE IF THE NUMBER IS OUT OF RANGE)
S        printf ("The number %d is not between 1 and 50\n",number);

PV    CHANGE (ADD PARENTHESES AROUND FOR AND WHILE CONDITIONS
                TO OVERRIDE DEFAULT PRECEDENCE)
S        if ((number < 1) || (number > 50))
```

```
S           } while ((number < 1) || (number > 50));

PV    CHANGE (MOVE STATEMENT THAT ECHOS BACK THE INPUT,
              TO AFTER DO WHILE LOOP)

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    RUN (PROG)

P     READ (SCREEN - A NUMBER BETWEEN 1 AND 50)
P     TEST (2, IT ACCEPTED THAT ONE)
PV    TEST (60, TELLS ME IT'S NOT BETWEEN, ECHOS, AND ASKS ME AGAIN)
PV    TEST (-1, TELLS ME SAME THING)
P     COMMENT (IT WORKS GREAT)

TIME 1:10:00 - TASK #3

PV    EDIT (PROG.C)
PV    CHANGE (ADD AN EXTRA LINE FEED BEFORE THE PROMPT)
S        printf("\nInput a number between 1 and 50: ");

P     NEED (TO DO SOMETHING UNTIL I FIND THE NUMBER)
P     COMMENT (COULD DO A WHILE FOREVER, THEN BREAK WHEN I FIND IT)
PV    INSERT (WHILE LOOP)
S        while (1)  {
S          }

P     NEED (TO DO SOME INITIALIZING)
PV    GOTO (ABOVE WHILE LOOP)
PV    INITIALIZE (I) FOR THE ARRAY INDEX
S        i=10;          /* start index in middle of array */

PV    GOTO (INSIDE OF WHILE LOOP)
P     NEED (TO COMPARE MY NUMBER TO THE ARRAY)
P     INSERT (IF STATEMENT)
S        if (nums[i] == number)      /* found number - done */

P     NEED (TO KEEP TRACK OF WHETHER I FOUND IT OR DIDN'T FIND IT)

P     READ (INSTRUCTIONS FOR TASK 3)
P     COMMENT (I DON'T NEED TO KEEP TRACK OF WHICH ONES IT'S BETWEEN)

PV    NEED (TO BREAK AND CHECK AT END IF I'VE FOUND IT OR NOT)
PV    NEED (TO PRINT HERE, SINCE THIS IS THE ONLY PLACE I'M LIKELY TO DECIDE THAT
            I'VE FOUND THE NUMBER)

PV    PRINT (FOUND MESSAGE) TO SCREEN
S      printf ("Input number, %d, found at array index %d\n",number,i);
PV    INSERT (BREAK THE LOOP AT THAT POINT, SINCE I'M DONE)
S        break;

PV    CHANGE (ADD BRACKETS AROUND STATEMENTS UNDER THE IF)
P     COMMENT (IF I DIDN'T FIND IT, NEED TO SEE IF IT'S > OR <)

PV    INSERT (IF STATEMENT)
S        if (nums[i] < number)
P     COMMENT (COMPARE IT TO THE LEFT HALF)

P     COMMENT (A RECURSIVE FUNCTION MIGHT WORK BETTER)
P     COMMENT (CALL IT ONCE FROM THE MAIN PROGRAM)
P     COMMENT (REPEAT SEARCH, WITH A NEW START AND END)
P     COMMENT (DO IT FOR THE LEFT HALF OR THE RIGHT HALF,
               BREAKING IT DOWN UNTIL ONLY COMPARING 1 NUMBER)

P     COMMENT (MY FUNCTION WILL RETURN EITHER A -1 IF IT DOESN'T FIND
               IT, OR THE ARRAY INDEX IF IT DOES)

PV    GOTO (JUST BELOW ECHO BACK OF INPUT NUMBER)
PV    INSERT (I = BINARY SEARCH FUNCTION)
PV    INSERT (PARAMETERS: ARRAY, START=1, END=20, SEARCH FOR NUMBER)
S        i = bin_search(nums,1,20, number);

PV    INSERT (IF STATEMENT TO CHECK IF NOT FOUND)
PV    PRINT (NOT FOUND MESSAGE)
P     COMMENT (OTHERWISE, I FOUND THE RESULT)
PV    PRINT (SUCCESS MESSAGE)
S        if (i < 0)
S        printf("Input number, %d, does not exist in the array.\n");
S        else
S        printf("Input number, %d, found at array index %n\n", number, i);
```

TIME 1:20:00 - TASK #3

```
P     COMMENT (OOPS, DIDN'T PUT VARIABLE NAME IN PRINT STATEMENT ABOVE)
PV    CHANGE (ADD THE VARIABLE NAME, NUMBER)
S     printf("Input number, %d, does not exist in the array.\n",number);

P     COMMENT (THAT'S ALL MAIN NEEDS TO DO)

PV    DELETE (GET RID OF MY ORIGINAL LOGIC)

PV    DECLARE (INTEGER FUNCTION, BIN_SEARCH, WITH PARAMETERS
                ARRAY, START INDEX, STOP INDEX, AND NUMBER TO LOOK FOR)
S        int bin_search(array, start, stop, target)

PV    DECLARE (INTEGER POINTER, ARRAY)
PV    DECLARE (INTEGER, START, STOP, AND TARGET)
S        int *array, start, stop, target;

PV    INSERT (COMMENT)
S        /*  end of bin_search  */

PV    INSERT (IF STATEMENT TO TEST FOR EXIT (NOT FOUND) CONDITION)
S        if (start > stop)
S            return (-1);

PV    INSERT (IF STATEMENT TO TEST IF NUMBER FOUND)
S        if (start == stop)
S            return (start);
P     COMMENT (NO, IF START = STOP, COULD BE SEARCHING AN ARRAY OF 1 ELEMENT)

PV    CHANGE (IF START = STOP TEST IF TARGET = ARRAY[START])
PV    INSERT (IF FOUND, CAN RETURN THIS ARRAY INDEX)
PV    INSERT (ELSE RETURN A -1)
PV    INSERT (BRACKETS AROUND THAT JUST TO CLARIFY IT)
S        if (start == stop) {
S            if (target == array[start])
S                return(start);
S             else
S                return (-1);
S            }

P     COMMENT (ELSE I'VE GOT MORE THAT 1 ARRAY ELEMENT TO SEARCH)
P     NEED (TO COMPARE IT TO MIDDLE AND SEARCH LEFT OR RIGHT HALF)

P     COMMENT (COMPUTE THE MIDDLE FIRST)
PV    CALCULATE (MIDDLE = HALF WAY BETWEEN START AND STOP)
S        middle = (start + stop) / 2;
PV    DECLARE (INTEGER, MIDDLE)
S        int middle;

PV    INSERT (IF STATEMENT FOR LESS THAN CASE)
P     COMMENT (WHAT ARE MY ARGUMENTS FOR THIS THING?)
P     NEED (TO RETURN SEARCH OF SAME ARRAY, SEARCHING LEFT HALF)
P     COMMENT (SO START AT SAME PLACE, AND STOP VALUE IS ONE LESS
                THAN THE MIDDLE ONE, STILL WANT TO LOOK FOR TARGET)
S        if (target < array[middle])      /* search left half */
S                   return(bin_search(array, start, middle-1, target));

PV    INSERT (IF STAEMENT TO TEST IF EQUAL TO ARRAY[MIDDLE])
PV    INSERT (RETURN MIDDLE)
S        if (target == array[middle])  /* found target */
S                   return(middle);
```

TIME 1:30:00 - TASK #3

```
P     COMMENT (ELSE IT MUST BE > MIDDLE, SO SEARCH THE RIGHT HALF)
P     COMMENT (RIGHT IS SIMILAR TO WHAT I DID IN THE LEFT HALF)
PV    INSERT (ELSE START AT MIDDLE+1, STOP AT SAME PLACE, SAME NUMBER)
S        else        /* search right half */
S           return(bin_search(array,middle+1, stop, target));

PV    GOTO (TOP OF BIN_SEARCH DECLARATION)
P     COMMENT (ECHO THE ARGUMENTS JUST TO TRACK HOW IT'S GOING)
PV    PRINT (START, STOP, AND TARGET) TO SCREEN
S        printf("in bin_search: start=%d, stop=%d, target=%d\n",
S                start, stop, target);
P     COMMENT (I'LL TAKE THAT OUT WHEN I'M DONE)

PV    EXIT (PROG.C)
```

```
PV    COMPILE (PROG.C)
PV    RUN (PROG)

P     COMMENT (LET'S TRY SOMETHING THAT'S THERE FIRST)
P     COMMENT (THIS WON'T WORK. WENT 1 TO 20. NEED TO GO FROM 0 TO 19)

P     COMMENT (LET'S SEE WHAT HAPPENS ANYWAY)
PV    TEST (ARRAY[10] IS 27. LET'S SEE IF IT FINDS THAT. IT BLEW UP)
V     ERROR (ACCESS VIOLATION)
P     READ (SCREEN - START=1, STOP=20, TARGET=27)

P     COMMENT (LET ME TRY FIXING THE STOP AND START BEFORE GOING ON)
PV    EDIT (PROG.C)

PV    CHANGE (ON FIRST BIN_SEARCH GO FROM 0 TO 19)
S        i = bin_search(nums,0,19, number);
PV    EXIT (PROG.C)

PV    COMPILE (PROG.C)
PV    RUN (PROG)

PV    TEST (MIDDLE IS 26, SO TRY THAT ONE.  SAME PROBLEM)
PV    ERROR (ACCESS VIOLATION)
P     READ (SCREEN - START=0, STOP=19, TARGET=26)
P     COMMENT (PROBLEM BEFORE IT CALLS THE NEXT BIN_SEARCH)

PV    EDIT (PROG.C)
P     COMMENT (MAYBE IT HAS THE WRONG ARRAY)

P     NEED (TO ADD A PRINT STMT AT TOP OF THE FUNCTION)
PV    PRINT (ARRAY[START] AND [STOP]) TO SCREEN
S        printf("in bin_search: array[start]=%d, array[stop]=%d\n"
S               array[start], array[stop];
PV    EXIT (PROG.C)

PV    COMPILE (PROG.C)
PV    ERROR (INSERTED A "," BEFORE IDENTIFIER "ARRAY")
P     COMMENT (LOOKS LIKE A COMMA'S MISSING)

PV    EDIT (PROG.C)
PV    CHANGE (ADD COMMA AT END OF LINE)
S        printf("in bin_search: array[start]=%d, array[stop]=%d\n",
S               array[start], array[stop];
PV    EXIT (PROG.C)

PV    COMPILE (PROG.C)
PV    ERROR (INSERTED A ")" BEFORE ",")
P     COMMENT (DIDN'T HAVE CLOSING PAREN, BUT IT COMPILED ANYWAY)

PV    EDIT (PROG.C)
PV    CHANGE (ADD CLOSING PARENTHESIS)
S        array[start], array[stop]);

TIME 1:40:00 - TASK #3

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)

PV    RUN (PROG)
PV    TEST (26.  START=0, STOP=19, TARGET=26.  ARRAY[START]=1, ARRAY[STOP]=50)
PV    ERROR (ACCESS VIOLATION)
P     COMMENT (GOT THE ARRAY OK, BECAUSE IT GOT THE FIRST AND LAST ONE)
P     COMMENT (BUT GOT A PROBLEM BEFORE BIN_SEARCH THE SECOND TIME)

P     HANDCHECK (HAD 0,19, AND 26.  WHAT ARRAY INDEX IS 26?  26 IS [9])
P     HANDCHECK (THE FIRST ONE I SHOULD LOOK AT IS 0+19...)
P     COMMENT (PUT MORE PRINTS IN THE FUNCTION TO SEE WHAT IT'S DOING)

PV    EDIT (PROG.C)
P     HANDCHECK (START IS 0, STOP IS 19, AND TARGET IS 26)
P     HANDCHECK (START > STOP IS FALSE, SO IT DOESN'T DO THAT)
P     HANDCHECK (START=STOP IS NOT TRUE, SO IT SHOULDN'T DO ANY OF THAT)

P     COMMENT (LET'S PRINT MIDDLE WHEN WE FIND IT)
PV    PRINT (MIDDLE) TO SCREEN
S        printf("bin_search: middle = %d\n", middle);

P     COMMENT (MIDDLE SHOULD BE 0+19 IS 19/2 IS 9.5 ROUNDS TO 9)
P     COMMENT (SHOULD = ARRAY[MIDDLE], SO IT SHOULD HAVE RETURNED MIDDLE
                AND EVERYTHING SHOULD HAVE WORKED)
```

```
P     COMMENT (SOMETHING'S WRONG.  PUT IN PRINTS TO SEE WHAT IT'S DOING)
PV    PRINT (IDENTIFY WHICH CASE IT'S USING) TO SCREEN
S        printf("target < array[middle]\n");
S        printf("target = array[middle]\n");
S        printf("target > array[middle]\n");

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    RUN (PROG)

PV    TEST (26, IT SET MIDDLE = 9,  IT SET TARGET = ...
P     COMMENT (FOUND THE EQUALS, THEN GOT AN ACCESS VIOLATION RETURNING)
P     COMMENT (LOOK AT WHAT GETS BACK, MAY BE THE WRONG VARIABLE TYPE)

PV    EDIT (PROG.C)
P     COMMENT (WHERE DO I CALL IT?)
P     READ (SOURCE -> SETTING I = EQUAL TO THE RETURN FROM THE FUNCTION)
P     COMMENT (I IS AN INTEGER AND FUNCTION'S DECLARED AS AN INTEGER)
PV    PRINT (THE INTEGER THAT IS RETURNED)
S        printf("main: bin_search returned %d\n", i);

P     COMMENT (OH, I SEE.  I'VE GOT A %N INSTEAD OF %D IN MY PRINT)
PV    CHANGE (%N TO %D)
S       printf("Input number, %d, found at array index %d\n",number, i);
PV    EXIT (PROG.C)
```

TIME 1:50:00 - TASK #3

```
PV    COMPILE (PROG.C)
PV    RUN (PROG)
PV    TEST (26, FOUND AT INDEX 9.  WONDERFUL)

P     COMMENT (TRY SOMETHING THAT'S AT AN END)
PV    TEST (1, FOUND AT INDEX 0)
P     COMMENT (TRY THE RIGHT END, TO MAKE SURE THE >'S WORK)
PV    TEST (50, FOUND AT INDEX 19)

P     COMMENT (TRY SOMETHING THAT'S NOT THERE)
PV    TEST (7, DOES NOT EXIST IN ARRAY)
P     READ (SCREEN -> CHECKING THE LOGIC FLOW)
P     COMMENT (LOOKS LIKE IT WORKS.  LET ME TAKE OUT THE DEBUG PRINTS)

PV    EDIT (PROG.C)
PV    DELETE (ALL DEBUG PRINT STMTS, EXCEPT INPUT NUMBER ECHO BACK)
P     COMMENT (LET'S MAKE SURE IT STILL RUNS)

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    RUN (PROG)

PV    TEST (9, FOUND AT INDEX 4)
PV    TEST (2, DOES NOT EXIST IN THE ARRAY)

P     COMMENT (PUT ONE MORE CHECK TO MAKE SURE WHEN IT'S DONE WITH THE
                SEARCH AND SAYS IT FOUND IT, IT REALLY DID FIND IT)
PV    EDIT (PROG.C)

P     COMMENT (IF IT DOES EXIST WE WANT TO MAKE SURE IT REALLY DOES)
PV    INSERT (IF STATEMENT TO CHECK IF NUMBER = NUMS[I])
P     COMMENT (ELSE I WANT TO PRINT THAT THERE IS AN ERROR IN MY LOGIC)
PV    PRINT (ERROR MESSAGE AND INDEX RETURNED OTHERWISE) TO SCREEN
S     if (number == nums[i])
S      printf("Input number, %d, found at array index %d\n", number, i);
S     else {
S         printf("ERROR - bin_search returned array index %d\n",i);
S         printf("        nums[%d] is actually %d, should be %d\n",
S                              i, nums[i], number);
S         }

PV    COMMENT (SO, IF WRONG ANSWER IS RETURNED, PROGRAM WILL CATCH IT)

PV    GOTO (END OF TASK 2)
PV    INSERT (CLOSE FILE)
S        fclose (fp);
PV    DELETE (CLOSE FILE STATEMENT AT THE END OF THE PROGRAM)
```

TIME 2:00:00 - TASK #3

```
PV    EXIT (PROG.C)
```

```
P     COMMENT (MAKE SURE THAT IT STILL WORKS)
PV    COMPILE (PROG.C)
PV    RUN (PROG)
PV    TEST (26.  FOUND AT INDEX 9)
P     COMMENT (OOPS, I DON'T NEED NUMBER =. NEED TO TAKE THAT OUT)
PV    TEST (2, DOES NOT EXIST IN ARRAY)

PV    EDIT (PROG.C)
PV    DELETE (PRINT THAT ECHOED BACK THE INPUT NUMBER)
PV    EXIT (PROG.C)

PV    COMPILE (PROG.C)
PV    RUN (PROG.C)

P     READ (SCREEN - NUMBER BETWEEN 1 AND 50)
PV    TEST (55, NOT BETWEEN 1 AND 50)
PV    TEST (-4, NOT BETWEEN 1 AND 50)
P     COMMENT (NUMBERS TOO BIG OR TOO SMALL - STILL PICKS THEM OUT)
P     TEST (1, FOUND.  50, FOUND.  34, NOT THERE)

P     COMMENT (I THINK I'M DONE)
```

SUBJECT #2 FINAL PROGRAM

```
#include <stdio.h>
main()
/*  This program initialises array of integers.  */
{
    FILE *fp;
    int i, sum=0;
    int number;
    int nums[20] = {1,3,5,8,9,12,13,19,25,26,27,29,33,35,38,39,40,43,45,50};
    printf ("The NUMS array has been initialised.\n");

/* 1st project */
    /* sum the array elements */
    for (i=0; i<20; i++) {
        sum += nums[i];
    }

    /* write the result to a file */
    fp = fopen("result.dat", "w+");
    fprintf(fp, "Array Sum = %d\n\n", sum);
    printf("The sum of the NUMS array is %d\n", sum);

/* 2nd project */
    /* print header to file */
    fprintf(fp, "Array \tDecimal\t Octal\nIndex \tRep     \t Rep\n");
    for (i=0; i<20; i++) {
        fprintf(fp,"%d \t %d \t %d\n", i, nums[i], conv_to_octal(nums[i]));
    }
    fclose (fp);
    printf("Octal conversion complete.\n");

/* 3rd project */
    do {
        /* get number from terminal */
        printf("\nInput a number between 1 and 50: ");
        scanf("%d", &number);

        /* make sure number between 1 and 50 */
        if ((number < 1) || (number > 50))
            printf ("The number %d is not between 1 and 50\n",number);
    } while ((number < 1) || (number > 50));

    /* find the number in array */
    i = bin_search(nums,0,19, number);
    if (i <0)
       printf("Input number, %d, does not exist in the array.\n",number);
    else {
        if (number == nums[i])
            printf("Input number, %d, found at array index %d\n", number, i);
        else {
            printf("ERROR - bin_search returned array index %d\n",i);
            printf("        nums[%d] is actually %d, should be %d\n", i,nums[i],number);
        }
    }
} /* end of main */

int bin_search(array, start, stop, target)
int *array, start, stop, target;
{
    int middle;
    if (start > stop)
        return (-1);
    if (start == stop) {
        if (target == array[start])
            return(start);
        else
            return (-1);
    }
    middle = (start + stop) / 2;
    if (target < array[middle]){                    /* search left half */
        return(bin_search(array, start, middle-1, target));
    }
    if (target == array[middle]) {                  /* found target */
        return(middle);
    } else {                                          /* search right half */
        return(bin_search(array,middle+1, stop, target));
    }
} /* end of bin_search */
```

```
/* convert integer argument to octal integer */
int conv_to_octal(arg)
int arg;
{
    int result = 0;
    int power_of_10 = 1;
    do {
        result += power_of_10 * (arg % 8);
        arg = arg / 8;
        power_of_10 *= 10;
    } while (arg > 0);
    return(result);
} /* end of conv_to_octal */
```

SUBJECT #3 DATA

TIME 0:00:00 - TASK #1

```
PV    EDIT (PROG.C)
P     READ (INSTRUCTIONS FOR TASK 1)
P     NEED (TO PUT IN SOME VARIABLES)

P     COMMENT (FIRST GET MY FILE READY)
PV    DECLARE (FILE POINTER, FP)
'        FILE *fp;

PV    INSERT (OPEN FILE, FILE.OUT, IN WRITE MODE)
PV    INSERT (TEST FOR ERRORS WHEN OPENING THE FILE)
PV    INSERT (ON ERROR, EXIT)
S        if ( (fp = fopen("file.out","w")) == -1)) {
S              perror("fopen");
S              exit(-1);
S            }

P     COMMENT (NOW SET UP MY PROGRAM)

PV    DECLARE (INTEGER, I) TO INDEX THROUGH MY ARRAY
PV    DECLARE (INTEGER, SUM) WHERE I ADD THEM UP
PV    INITIALIZE (SUM IN THE DECLARATION)
S     int i,sum=0,
S     nums[20]={1,3,5,8,9,12,13,19,25,26,27,29,33,35,38,39,40,43,45,50};

PV    INSERT (FOR LOOP)
PV    CALCULATE (SUM)
S        for(i=0;i<20];i++)
S             sum += nums[i];

PV    PRINT (SUM) TO FILE
S        fprintf(fp,"The sum of all the numbers is: %d.\n",sum);

P     COMMENT (I FORGOT THE LINE FEEDS)
PV    CHANGE (ADD SEVERAL NEWLINES)
S        fprintf(fp,"\n\n\nThe sum of all the numbers is: %d.\n",sum);

P     COMMENT (OUTPUT FILE SHOULD CLOSE AUTOMATICALLY)

P     READ (CODE, TO MAKE SURE LOGIC LOOKS OK)
P     COMMENT (I HAVE A TYPO ON MY FOR LOOP)
PV    CHANGE (DELETE BRACKET)
S        for(i=0;i<20;i++)

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    ERROR (UNEXPECTED ")" IGNORED)

PV    RUN (PROG)
PV    TYPEFILE (FILE.OUT)
P     COMMENT (IT GOT 500)

P     HANDCHECK (ADD ARRAY NUMBERS ON THE CALCULATOR)
P     COMMENT (LOW AND BE"OLD: 500)
```

TASK 2

```
P     READ (INSTRUCTIONS FOR TASK 2)
P     COMMENT (I WILL USE THE DIVISION TECHNIQUE)

PV    EDIT (PROG.C)
P     CHANGE (COMMENT OUT LINES THAT PRINTED THE SUM)
S       /* fprintf(fp,"\n\n\nThe sum of all the numbers is: %d.\n",sum); */
```

TIME 0:10:00 - TASK #2

```
P     COMMENT (SHOULD I USE A STRING ARRAY TO STORE MY VALUES?)
P     COMMENT (WHILE I'M THINKING, I'M GOING TO FORMAT MY OUTPUT)

P     NEED (TO LEAVE THE ARRAY SUM IN THERE)
PV    CHANGE (DELETE COMMENTS MARKERS AROUND SUM OUTPUT)
PV    CHANGE (ADD MORE LINE FEEDS AT END OF PRINT)
S        fprintf(fp,"\n\n\nThe sum of all the numbers is: %d.\n\n\n",sum);

PV    PRINT (ARRAY, DECIMAL, OCTAL, LINEFEED) TO FILE
PV    PRINT (INDEX, REP, REP, 2 LINE FEED) TO FILE
```

```
S       fprintf(fp,"Array        Decimal     Octal\n");
S       fprintf(fp,"Index         Rep         Rep\n\n");

P     COMMENT (I'LL CALCULATE THE THINGS AS I PRINT THEM OUT)
P     COMMENT (WHAT VARIABLES DO I NEED?)

P     COMMENT (ALL NUMBERS ARE 2 DIGITS, SO JUST MAKE IT 2 CHARACTERS)
P     COMMENT (COUNT OFF TO GET APPROXIMATELY WHERE I NEED TO BE)
P     COMMENT (I MIGHT HAVE SOME THAT ARE 3 LONG)

P     COMMENT (50 DECIMAL IS 62 OCTAL, SO I JUST NEED 2)
PV    PRINT (DECIMAL, DECIMAL) TO FILE
S       fprintf(fp,"  %2d          %2d\n",i,nums[i]);

P     NEED (TO FIGURE OUT MY OCTAL REPRESENTATION)
P     COMMENT (I COULD PUT IT IN A STRING)
P     COMMENT (PUT NUMBER MOD 8 AS MY FIRST CHARACTER)

P     COMMENT (IF I USE THIS TECHNIQUE, KEEP INCREMENTING CHARACTER COUNT, MAKE A SMALL
               ARRAY, AND KEEP DOING MODS.  SAVE RESULT AND REMAINDER, PUT REMAINDER IN A
               CHARACTER, BUMP CHARACTER OVER 1 AND DIVIDE AND JUST KEEP THE DIVIDING)

P     COMMENT (PUT IT INTO A STRING, OR DO THIS IS BY SHIFTING)
P     COMMENT (USE A CHARACTER STRING)

PV    CHANGE (ADD STRING TO OUTPUT STATEMENT)
S       fprintf(fp,"  %2d          %2d           %s\n",i,nums[i],str);

P     NEED (TO REMEMBER TO NULL TERMINATE MY STRING)
P     READ (INSTRUCTIONS FOR TASK 2)

P     COMMENT (937 IS 1651 OCTAL.  THE 6 WOULD GO FIRST)
P     COMMENT (SO IF I BUMP MY CHARACTERS...)
P     COMMENT (DO A MOD 8 FIRST TO GET THE LEAST DIGIT)
P     COMEMNT (DO IT WITH A CHARACTER STRING)
P     COMMENT (CONVERT AND CONCATENATE VALUES ONTO END OF STRING)
P     COMMENT (THAT WOULD WORK)

P     NEED (TO SAVE THIS TO A STRING)
PV    DECLARE (CHARACTER ARRAY OF 10, STR)
S       char str[10];

P     NEED (ANOTHER CHARACTER STRING)
P     COMMENT (PROBABLY 2 IS ALL WE'LL NEED...)
PV    DECLARE (CHARACTER ARRAY OF 2, TMP)
S       char str[10],tmp[2];

TIME 0:20:00 - TASK #2

PV    INSERT (FOR LOOP AROUND PRINT STATEMENT)
S       for (i=0; i<20: i++) {
S             }

P     COMMENT (HOW SHOULD I DO IT?)
P     NEED (A COUPLE MORE VARIABLES)

PV    GOTO (DECLARATIONS)
PV    DECLARE (INTEGER, J)
S       int i,j,sum=0,
S        nums[20]={1,3,5,8,9,12,13,19,25,26,27,29,33,35,38,39,40,43,45,50};

PV    GOTO (INSIDE FOR LOOP)
PV    CALCULATE (J)
S       j = nums[i] % 8;

PV    INSERT (READ INPUT INTO TMP)
S       sprintf(tmp,"%d",j);

P     COMMENT (THIS GIVES ME THE LEAST SIGNIFICANT DIGIT)
P     COMMENT (KEEP CATTING THE NUMBER ONTO THE END OF WHAT I GET)

PV    INITIALIZE (STR) TO NULL TERMINATE IT
S       str[0] = '\0';

PV    INSERT (STRING CONCATENATE TEMP ON END OF STRING)
S       strcat(str,tmp);

PV    INSERT (WHILE LOOP)
S       while (j > ) {
S             }
```

```
P     COMMENT (IF I MAKE IT > 0, WILL THAT LAST CASE WOULD GET DONE?)
P     COMMENT (NOT SURE WHAT A MOD WILL DO WHEN A NUMBER IS LESS THAN 8)

PV    CHANGE (CONDITION IN WHILE LOOP)
S        while (j > 7) {
S                }

P     COMMENT (TAKE IT AS A SPECIAL CASE FOR NOW)
PV    GOTO (INSIDE WHILE LOOP)
PV    CALCULATE (J)
S        j = number % 8;
P     COMMENT (THIS ISN'T GOING TO WORK)

PV    GOTO (ABOVE WHILE LOOP)
PV    INITIALIZE (J)
S        j = nums[i];

PV    GOTO (INSIDE WHILE LOOP)
PV    CHANGE (J CALCULATION)
S        j = j%8;

P     NEED (TO RUN THIS PROGRAM AND SEE WHAT IT RETURNS)

P     COMMENT (IN LAST CASE, MIGHT HAVE TO DUPLICATE THESE LINES)
P     COMMENT (COULD DO IT INSIDE THE LOOP AND BREAK OUT OF THE LOOP THEN JUST CHECK
                DOWN AT THE BOTTOM)

P     COMMENT (IF J < 8 THEN I TOOK CARE OF THE LAST TIME AND NOT MAKE J GO AROUND AGAIN)
P     COMMENT (THAT WOULD WORK, BUT THIS SHOULD GET ME ALMOST DONE)

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
P     COMMENT (SEE IF I'VE MADE ANY TYPOS)

PV    ERROR (UNEXPECTED ")" IGNORED)
PV    ERROR (TMP NOT DECLARED WITHIN SCOPE OF THIS USAGE)
PV    ERROR (STR NOT DECLARED WITHIN SCOPE OF THIS USAGE)
PV    ERROR (FOUND ":" WHEN EXPECTING ARITHMETIC OPERATOR)

PV    EDIT (PROG.C)

PV    CHANGE (MOVE CHARACTER DECLARATIONS ABOVE FIRST PRINT STMT)
PV    CHANGE (DECLARATION OF TMP TO ARRAY OF 3)
S        char str[10], tmp[3];

PV    READ (MANUAL ON STRING CAT)
P     NEED (TO SEE WHETHER I NEED TO INCLUDE A .H FILE HERE)
P     COMMENT (DOESN'T LOOK I NEED ANY SPECIAL INCLUDE FILE FOR STRING CAT)

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    ERROR (UNEXPECTED ")" IGNORED)
PV    ERROR (FOUND ":" WHEN EXPECTING ARITHMETIC OPERATOR)
PV    EDIT (PROG.C)

PV    GOTO (INSIDE FOR LOOP)
P     COMMENT (PUT A COLON WHERE I NEED A SEMICOLON)
PV    CHANGE (: AFTER I<20 TO ;)
S        for (i=0; i<20; i++)

TIME 0:30:00 - TASK #2

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    ERROR (UNEXPECTED ")" IGNORED)

PV    RUN (PROG)
PV    TYPEFILE (FILE.OUT)

P     NEED (TO RE-INITIALIZE MY STRING EACH TIME)
P     COMMENT (THE LAST DIGIT IS GETTING TAKEN CARE OF)
P     COMMENT (I THINK THIS IS GONNA WORK)

PV    EDIT (PROG.C)
PV    CHANGE (MOVE STR INITIALIZATION INSIDE FOR LOOP)
S        for (i=0; i<20; i++) {
S             str[0] = '\0';

PV    COMMENT (BACK TO THE PROBLEM OF THE LAST LINE TYPE ERROR)
```

```
PV   CHANGE (WHILE LOOP CONDITION SO IT LOOPS FOREVER)
S       while (1) {
S            }

PV   GOTO (END OF WHILE LOOP)
PV   INSERT (IF STATEMENT FOR SPECIAL CASE)
S       if (j < 8)
S            break;

P    READ (MANUAL ON BREAK)
P    COMMENT (NO...I'LL DO IT ANOTHER WAY)

PV   CHANGE (IF STATEMENT SET J=0 INSTEAD OF BREAK-ING)
S               if (j < 8)
S                    j = 0;

P    COMMENT (CHECK FOR J = 0)
PV   CHANGE (WHILE LOOP CONDITION FROM FOREVER)
S       while (j != 0) {

PV   EXIT (PROG.C)
P    COMMENT (ONCE I GET CLOSE, I USE TRIAL AND ERROR)
PV   COMPILE (PROG.C)
PV   ERROR (UNEXPECTED ")" IGNORED)
P    ERROR (I FORGOT TO FIX THAT ONE WARNING)

PV   RUN (PROG)
PV   TYPEFILE (FILE.OUT)
P    COMMENT (I'M STILL OFF BY ONE.  I'M MISSING MY FIRST CHARACTER)
P    COMMENT (MY STRING CONCAT IS NOT WORKING)

P    HANDCHECK (CONVERT ON CALCULATOR TO MAKE SURE LAST NUMBERS ARE CORRECT)
P    HANDCHECK (25 SHOULD BE 1. THAT'S CORRECT)
P    HANDCHECK (26 SHOULD BE 2. OK)

P    COMMENT (I'M JUST MISSING THE FIRST CHARACTER)

PV   EDIT (PROG.C)

P    NEED (TO FIGURE OUT WHAT J MOD 0 IS GONNA GET ME)
P    COMMENT (I THINK A MOD IS GONNA GIVE ME 0)

PV   DELETE (IF STATEMENT THAT SETS J TO ZERO)

P     HANDCHECK (FIRST TIME THROUGH, FOR 1 IT WOULD GET 1 AND IT WOULD GO THROUGH AND
J...)

P    COMMENT (LOOKING AT THE MOD.  I NEED THE ACTUAL RESULT ALSO)
P    NEED (ANOTHER VARIABLE)

PV   DECLARE (INTEGER, MOD)
PV   DELETE (DECLARATION OF J)
PV   DECLARE (INTEGER, DIV)
S       int i,mod,div,sum=0,
S         nums[20] = {1,3,5,8,9,12,13,19,25,26,27,29,33,35,38,39,40,43,45,50};

PV   GOTO (INITIALIZATION OF J, ABOVE WHILE LOOP)
PV   CHANGE (USE DIV IN PLACE OF J)
S       div = nums[i];

PV   CHANGE (WHILE CONDITION TO USE DIV INSTEAD OF J)
S       while (div != 0) {

PV   GOTO (INSIDE WHILE LOOP)
PV   CHANGE (USE DIV INSTEAD OF J IN J CALCULATION)
S       j = div%8;

PV   GOTO (END OF WHILE LOOP)
PV   CALCULATE (DIV)
S       div /= 8;

P    COMMENT (FIND REMAINDER AND GET THE VALUE)

PV   CHANGE (WHILE CONDITICN FROM != TO >)
S       while (div > 0) {

P    COMMENT (IF THIS DOESN'T WORK, I'LL ADD PRINT STATEMENTS
              TO SEE HOW MANY TIMES I'M LOOPING AND WHAT MY VALUES ARE)
```

```
PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    ERROR (UNEXPECTED ")" IGNORED)
PV    ERROR (J NOT DECLARED WITHIN THE SCOPE OF THIS USAGE)

P     NEED (TO CHANGE ALL MY TERMS WITH A J TO BE MOD)
PV    EDIT (PROG.C)

PV    CHANGE (J'S TO MOD'S)
S        mod = div%8;
S        sprintf(tmp,"%d",mod);
PV    EXIT (PROG.C)

TIME 0:40:00 - TASK #2

PV    COMPILE (PROG.C)
PV    ERROR (UNEXPECTED ")" IGNORED)
P     COMMENT (STILL HAVE TO TAKE CARE OF PARENTHESES PROBLEM)

PV    RUN (PROG)
PV    TYPEFILE (FILE.OUT)
P     COMMENT (THE NUMBERS ARE RIGHT, I JUST HAVE THEM BACKWARD)
P     COMMENT (I JUST DID MY CON CAT WRONG)
P     COMMENT (CHECK THEM EVEN THOUGH THEY'RE BACKWARDS)
P     HANDCHECK (50 IN OCTAL IS 62)

PV    EDIT (PROG.C)
P     COMMENT (MAYBE IT'S NOT EVEN A STRING CAT I WANT TO DO)

PV    CHANGE (SWITCH ORDER OF STR AND TMP, TO PUT STRING ON END OF TEMP)
PV    INSERT (THEN COPY TEMP RIGHT BACK INTO STRING, TO SWAP IT AROUND)
S        strcat(tmp,str);
S        strcpy(str,tmp);

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    ERROR (UNEXPECTED ")" IGNORED)
PV    RUN (PROG.C)

PV    TYPEFILE (FILE.OUT)
P     COMMENT (LOOKS GOOD)
P     HANDCHECK (CONVERSIONS)
P     COMMENT (PART B IS DONE)

TASK 3

P     READ (INSTRUCTIONS FOR TASK 3)
PV    EDIT (PROG.C)

P     CHANGE (COMMENT OUT THE STUFF WE'VE DONE SO FAR)
S        /*  for(i=0;i<20;i++)
S            (CODE FOR TASKS 1 AND 2)
S        */

PV    GOTO (TOP OF PROGRAM)
P     NEED (TO PROMPT THE USER)
P     READ (INSTRUCTIONS)

PV    INSERT (PROMPT)
S        printf("\n\nPlease enter a number between 1 and 50:");

PV    INSERT (READ INPUT NUMBER AND THE CARRIAGE RETURN)
S        gets(str);
PV    INSERT (PUT INTEGER PART OF INPUT IN VARIABLE I)
S        sscanf(str,"%d",&i);

P     READ (INSTRUCTIONS FOR TASK 3)
P     NEED (AN INDEX)

PV    GOTO (DECLARATIONS)
PV    DECLARE (INTEGER, INDEX)
PV    INITIALIZE (INDEX)
S        int i,index = 0,mod,div,sum=0,
S          nums[20] = {1,3,5,8,9,12,13,19,25,26,27,29,33,35,38,39,40,43,45,50};

PV    GOTO (AFTER INPUT CODE)
P     COMMENT (ASSUME IT'S BETWEEN 0 AND 50)

P     COMMENT (NOW LET'S SEE IF I CAN FIND IT)
PV    NEED (TO COMPARE NUMBER TO THE MIDDLE ENTRY IN THE ARRAY)
```

```
P     NEED (TO BE IN A LOOP)
P     READ (INSTRUCTIONS FOR TASK 3)
P     COMMENT (HOW WILL I KNOW THAT I'M DONE?)

P     COMMENT (KEEP AN INDEX)
P     COMMENT (IF IT WAS >, THEN CHECK BETWEEN 10 AND 20)
P     COMMENT (HOW AM I GONNA KNOW THE DIFFERENCE?)

P     NEED (TO ADD HALF THE NUMBER OR SUBTRACT HALF THE NUMBER...)
P     HANDCHECK (IF IT WAS > 10, TAKE HALF OF 10, AND ADD IT TO GET 15)
P     HANDCHECK (IF IT WAS < 10, TAKE HALF OF 10, IS 5, AND SUBTRACT IT)
P     HANDCHECK (NOW CHECK THE 5.  FOLLOW IT ONE MORE STEP FURTHER)

P     HANDCHECK (IF I HAD PICKED 10 AND IT WAS HIGHER NOW I PICK 15)
P     HANDCHECK (NOW IF IT WAS STILL HIGHER, TAKE HALF THAT NUMBER)
P     COMMENT (NO THAT WOULDN'T WORK...)

TIME 0:50:00 - TASK #3

P     COMMENT (TAKE DIFFERENCE OF THE NUMBERS AND HALF THAT NUMBER...)
P     HANDCHECK (20 AND 10.  DIFFERENCE AND HALF IT WOULD BE 15)
P     COMMENT (SO I KNOW IT'S EITHER ADDED OR SUBTRACTED)

P     COMMENT (IF I ADDED ONE STEP FURTHER...)
P     HANDCHECK (DIFF BETWEEN 15 AND 10, DIVIDED BY 2, I WOULD GET 2,
                 SO I WOULD CHECK +2 OR -2)

P     NEED (TO USE PENCIL AND PAPER AND FOLLOW THROUGH THE INDEXES)
P     HANDCHECK (USE ALL THE NUMBERS BETWEEN 1 AND 12)
P     HANDCHECK (FIRST TIME, INDEX = TOTAL NUMBER = 12)
P     HANDCHECK (CHECK 6.  SAY I'M LOOKING FOR THE NUMBER 10)
P     HANDCHECK (INDEX[6] = 6.  I CHECK MY TARGET AND IT'S >)
P     HANDCHECK (SO NUMBER'S BETWEEN INDEX[6] AND [1])
P     HANDCHECK (INDEXES GO FROM 0 TO 11)
P     HANDCHECK (IF I TOOK DIFFERENCE OF LAST TWO NUMBERS...)
P     HANDCHECK (DIFFERENCE WOULD BE 5/2, GIVES ME 2.  SO I'D ADD 2)
P     HANDCHECK (NOW CHECK 8.  NO, 8'S NOT RIGHT)

P     COMMENT (IF I WANT TO GO BETWEEN 6 AND 8, DO I WANT TO CHECK BETWEEN 6 AND 8?)

P     HANDCHECK (FIRST TIME I'D BE ON 6.  THEN 3, THEN PROBABLY 1)
P     COMMENT (ASSUME I DIVIDE, IT WILL BE A FACTOR OF 2 EACH TIME)
P     HANDCHECK (FIRST TIME LOOK AT 6.  IF IT WAS ABOVE 6, ADD 3)
P     HANDCHECK (IF IT WAS ABOVE AGAIN, ADD 1)
P     HANDCHECK (IF THAT WASN'T IT, THEN THAT WOULDN'T HAVE WORKED)

P     COMMENT (TRY A DIFFERENT SCENARIO)
P     HANDCHECK (WE HAVE 12 ENTRIES.  LOOKING FOR THE NUMBER 6)
P     HANDCHECK (NEED TO INDEX OUR VALUE.  IT EQUALS THE NUMBER/2 - 1)
P     HANDCHECK (THAT GIVES ME 5 AS MY FIRST VALUE)

P     READ (INSTRUCTIONS FOR TASK 3)
P     COMMENT (I'M GONNA HAVE TO FIGURE OUT AN ALGORITHM)
P     COMMENT (PROBLEM IS THAT YOU GET DOWN TO BETWEEN 2 OR 3 NUMBERS,
               AND THERE'S 3 NUMBERS IN BETWEEN.  SUBTRACT 2 OR 1?)

P     COMMENT (THIS NEEDS TO BE WELL THOUGHT OUT BEFORE I START)
P     COMMENT (SOMETIMES IT HELPS TO THINK OF SOMETHING ELSE)

TIME 1:00:00 - TASK #3

P     HANDCHECK (WRITE TEST LIST ON PAPER)
P     HANDCHECK (GO FROM 1 TO 19.  LOOK FOR 4)
P     HANDCHECK (GOES OUT TO 10, AND GETS 15.  IT'S <)
P     HANDCHECK (INDEX = 10 - 10/2.  SO INDEX = 5)
P     HANDCHECK (CHECK 5. IT'S < 5. SO INDEX=INDEX-INDEX/2. WILL BE 3)
P     HANDCHECK (CHECK 3. IT'S STILL < 5.  I'M GONNA SAY...)
P     COMMENT (WAIT, IT SHOULD BE (INDEX + 1)/2...)

P     HANDCHECK (INDEX = 10.  INDEX - ((INDEX + 1)/2) WOULD GIVE US 5)
P     HANDCHECK (THEN 5-(6/2)  WILL GIVE ME 3.  SO CHECK 3)
P     HANDCHECK (NEXT TIME SAY 3-(4/2). SO CHECK 1)
P     HANDCHECK (THEN 1-(2/2).  GOT 0, SO THAT WOULDN'T WORK)

P     COMMENT (CHECK IF THERE'S A DIFFERENCE OF 1 BETWEEN THE NUMBERS?)

P     HANDCHECK (FIRST TIME INDEX = 10.  20-(20+1)/2 GETS 10)
P     HANDCHECK (LOOK AT 10, NUMBER WE'RE LOOKING FOR IS <)
P     HANDCHECK (SO INDEX=INDEX-((INDEX +1)/2).  GET INDEX OF 5)
```

```
P     HANDCHECK (LOOK AT 5. IT'S HIGHER THAN 5. SO IT'S 5+((5+1)/2))
P     HANDCHECK (LOOK AT 8. IT'S LOWER THAN 8. THIS ISN'T GONNA WORK...)

P     NEED (TO KEEP TRACK OF OLD INDEX.  OLD HIGH AND OLD LOW...)
P     HANDCHECK (FIRST CASE: LOW = 0, HIGH = 19, MIDDLE = 10)
P     HANDCHECK (IF IT'S HIGHER, LOW=10, HIGH=20, MIDDLE=(10+19)/2=14)
P     HANDCHECK (IF IT'S >, WE CAN KEEP GOING UNTIL LOW = HIGH)

P     HANDCHECK (WRITE #'S ON PAPER FOR HAND CALCULATIONS.  LOOK FOR 13)
P     HANDCHECK (LOW = 0, HIGH = 19. (0 + 19)/2...START ON 9)
P     HANDCHECK (LOOK AT 9. > 9, SO LOW=9, HIGH=19, MID=19+9/2=28/2=14)
P     HANDCHECK (LOOK AT 14. < 14, SO HIGH=14, LOW STAYS 9, AND MID=11)
P     HANDCHECK (LOOK AT 11. > 11, SO LOW=11, HIGH STAYS 14,MID=25/2=12)
P     HANDCHECK (LOOK AT 12. > 12, SO LOW=12, HIGH STAYS 14, MID=13)
P     COMMENT (WE HAVE FOUND OUR ANSWER)

P     COMMENT (LOOKS LIKE THIS WOULD WORK FOR ABOUT ANY CASE)

P     COMMENT (LET'S TRY ANOTHER CASE)
P     HANDCHECK (LOW = 0, HIGH = 19, MID = 10.  TRY 10. LOWER THAN 10)
P     HANDCHECK (LOW = 0, HIGH = 10, MID = 5.   TRY 5.  LOWER THAN 5)
P     HANDCHECK (LOW = 0, HIGH = 5,  MID = 2.   TRY 2.  LOWER THAN 2)
P     HANDCHECK (LOW = 0, HIGH = 2,  MID = 1.   TRY 1.  LOWER THAN 1)
P     HANDCHECK (LOW = 0, HIGH = 1,  MID = 0.   TRY 0.  FOUND IT)

P     COMMENT (HOW WOULD WE KNOW THAT WE'RE DONE THEN?)
P     COMMENT (WHEN HIGH = LOW.  I THINK THAT'S GONNA WORK)

TIME 1:10:00 - TASK #3

P     COMMENT (IF LOWER THAN 0, THEN HIGH=0. THEN DOES HIGH = LOW? YES)
P     COMMENT (THEN IT WASN'T FOUND)

PV    DECLARE (INTEGERS, LOW, MID, AND HIGH)
S        int low,mid,high,i,index = 0,
S          mod,div,sum=0,
S          nums[20] = {1,3,5,8,9,12,13,19,25,26,27,29,33,35,38,39,40,43,45,50};

PV    INSERT (WHILE LOOP)
S        while( (low != high) && (nums[mid] != i)) {

PV    GOTO (ABOVE WHILE LOOP)
PV    INITIALIZE (LOW AND HIGH)
S        low = 0;
S        high = 19;

PV    CALCULATE (FIRST MID)
S        mid = (high + low)/2;

PV    GOTO (INSIDE WHILE LOOP)
PV    INSERT (IF STATEMENT TO RESET LOW)
PV    INSERT (ELSE CLAUSE TO RESET HIGH)
S        if (i > nums[mid] )
S            low = mid;
S        else
S            high = mid;

PV    CALCULATE (MID)
S        mid = (high + low)/2;

P     NEED (PRINT STATEMENT TO TELL ME WHAT'S GOING ON)
PV    GOTO (BEGINNING OF WHILE LOOP)
PV    PRINT (LOW, MID, HIGH)
S        printf("\nlow = %d;  mid = %d;  high = %d",low,mid,high);

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    ERROR (UNEXPECTED ")" IGNORED)
P     COMMENT (STILL HAVE MISMATCHED PARENTHESES)
PV    RUN (PROG)

PV    TEST (13. LOW, MID, HIGH ARE 0, 9, 19.  THEN 0, 4, 9)
PV    COMMENT ( SOMETHING'S WRONG. LET ME TRY A DIFFERENT NUMBER)

PV    TEST (0, WENT TO 0,9,19, THEN 0,4,9, THEN 0,2,4, THEN 0,1,2,
              THEN 0,0,1)
PV    TEST (THE HIGHEST, 50.  INFINITE LOOP AT 18,18,19)

PV    EDIT (PROG.C)
PV    INSERT (IF NUMS[MID] = I)
```

```
PV    NEED (TO PRINT OUT RESULTS)

TIME 1:20:00 - TASK #3

PV    GOTO (AFTER WHILE LOOP)
PV    INSERT (IF STATEMENT TO CHECK IF NUMBER FOUND)
PV    PRINT (FOUND MESSAGE, I , AND MID)
S       if (numb[mid] == i)
S         printf("\nInput number, %d, found at array index %d.\n",i,mid);

PV    INSERT (ELSE CLAUSE, FOR NOT FOUND CASE)
PV    PRINT (NOT FOUND MESSAGE)
S       else
S         printf("\nInput number, %d, does not exist in the array.\n");

P     COMMENT (ACTUALLY THE WAY I WAS FIGURING IT OUT IT WOULD BE +1)
PV    CHANGE (ADD ONE TO MID INITIALIZATION)
S       mid = (high + low + 1)/2;

P     COMMENT (LET ME DOUBLE CHECK MY VALUES)
P     HANDCHECK (FOR 0, WE WOULD GET 9.  HIGH = 9.  LOW = 4)
P     HANDCHECK (LOOK AT 4 AND WE GET 2.  AND THEN WE GET 1)
P     COMMENT (NOW IT'S BETWEEN 0 AND 1. THAT WOULDN'T WORK...)

P     HANDCHECK (IF MID WAS 1 AND IT WAS STILL LOWER THAT, BECAME 1 NOW)
P     HANDCHECK (FIND OUR MID.  (HIGH + LOW)/2)
P     COMMENT (DOESN'T WORK ON THAT SIDE.  BUT THE OTHER WAY IT DOES)

P     HANDCHECK (THAT WOULD BE 10.  11 GIVES US 5. 5+1 GIVES US 3)
P     HANDCHECK (THAT GOES 3, THAT GOES 2, AND THAT GOES TO 1)
P     COMMENT (JUST 1 AGAIN.  EVIDENTLY STUCK IN A LOOP)

PV    CHANGE (REMOVE PLUS ONE FROM MID INITIALIZATION)
S       mid = (high + low)/2;

P     COMMENT (I HAVE TO FIGURE OUT SOME SORT OF COMPARISON)
P     COMMENT (RUN IT AND LOOK AT THIS OUTPUT SO I CAN FIGURE IT OUT)
PV    EXIT (PROG.C)

PV    COMPILE (PROG.C)
PV    ERROR (UNEXPECTED ")" IGNORED)
PV    ERROR (NUMB NOT DECLARED WITHIN SCOPE OF THIS USAGE)
P     COMMENT (I MADE A MISTAKE - TYPO)

PV    EDIT (PROG.C)
PV    CHANGE (NUMB TO NUMS)
S       if (nums[mid] == i)

PV    COMMENT (FORGOT TO PUT THE INDEX ON THE END OF THIS PRINT)
PV    CHANGE (ADD "I" TO NOT FOUND MESSAGE)
S       printf("\nInput number, %d, does not exist in the array.\n",i);

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
P     COMMENT (BUT I HAVEN'T FIGURED OUT HOW TO GET RID OF THIS BUG)
PV    ERROR (UNEXPECTED ")" IGNORED)
PV    RUN (PROG)

PV    TEST (1, FOUND AT INDEX 0)
PV    TEST (50, INFINITE LOOP)

PV    TEST (45, FOUND AT INDEX 18)
PV    TYPEFILE (PROG.C, TO SEE WHAT THE NUMBERS ARE)
PV    TEST (29, FOUND AT INDEX 11)

PV    TEST (ALL NUMBERS IN THE ARRAY)
PV    TEST (50, INFINITE LOOP WITH 18,18,19)

P     COMMENT (TRY SOME THAT AREN'T THERE)
PV    TEST (44, INFINITE LOOP WITH 17,17,18)
PV    TEST (2, INFINITE LOOP WITH 0,0,1)

TIME 1:30:00 - TASK #3

PV    EDIT (PROG.C)
PV    CHANGE (TAKE OUT EXTRA CLOSING PARENTHESIS)
S       if ( (fp = fopen("file.out","w")) == -1) {

PV    CHANGE (INITIALIZATION OF MID, FROM 10 TO 20)
S       mid = high;
```

```
PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    RUN (PROG)

PV    TEST (0, DOES NOT EXIST)
PV    TEST (1, FOUND AT 0)
PV    TEST (50, FOUND AT 19)

PV    TEST (46, INFINITE LOOP WITH 18,18,19)
P     COMMENT (ITS STUCK IN A LOOP WITH LOW = MID)

PV    EDIT (PROG.C)
PV    CHANGE (WHILE CONDITION FROM LOW != HIGH)
S        while( (low != mid) && (nums[mid] != i)) {

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    RUN (PROG)

PV    TEST (46, DOES NOT EXIST)
PV    TEST (1, FOUND AT 0)
PV    TEST (0, DOES NOT EXIST)
PV    TEST (44, DOES NOT EXIST)
PV    TEST (2, FOUND AT 2)

PV    EDIT (PROG.C)
PV    CHANGE (COMMENT OUT DEBUG PRINT STATEMENT)
S        /*       printf("\nlow = %d;  mid = %d;  high = %d",low,mid,high); */

PV    DELETE (COMMENT BRACKETS AROUND FIRST TWO TASKS)

PV    EXIT (PROG.C)
PV    COMPILE (PROG.C)
PV    RUN (PROG)

PV    TEST (46, DOES NOT EXIST)
PV    TYPEFILE (FILE.OUT)
P     COMMENT (LOOKS GOOD)
```

SUBJECT #3 FINAL PROGRAM

```
#include <stdio.h>

main()
/*  This program initializes array of integers.  */

{
    FILE *fp;
    int low,mid,high,i,index = 0,
        mod,div,sum=0,
        nums[20] = {1,3,5,8,9,12,13,19,25,26,27,29,33,35,38,39,40,43,45,50};
    char str[10], tmp[3];

    printf ("The NUMS array has been initialized.");

    if ( (fp = fopen("file.out","w")) == -1) {
       perror("fopen");
       exit(-1);
    }

    printf("\n\nPlease enter a number between 1 and 50:");
    gets(str);
    sscanf(str,"%d",&i);
    low = 0;
    high = 19;
    mid = high;
    while( (low != mid) && (nums[mid] != i)) {
/*       printf("\nlow = %d;  mid = %d;  high = %d",low,mid,high); */
       if (i > nums[mid] )
          low = mid;
       else
          high = mid;
       mid = (high + low)/2;
    }
    if (nums[mid] == i)
          printf("\nInput number, %d, found at array index %d.\n",i,mid);
       else
          printf("\nInput number, %d, does not exist in the array.\n",i);

    for(i=0;i<20;i++)
       sum += nums[i];
    fprintf(fp,"\n\n\nThe sum of all the numbers is: %d.\n\n\n",sum);
    fprintf(fp,"Array        Decimal     Octal\n");
    fprintf(fp,"Index         Rep         Rep\n\n");

    for (i=0; i<20; i++) {
       str[0] = '\0';
       div = nums[i];
       while (div > 0) {
          mod = div%8;
          sprintf(tmp,"%d",mod);
          strcat(tmp,str);
          strcpy(str,tmp);
          div /= 8;
       }
       fprintf(fp,"  %2d          %2d           %s\n",i,nums[i],str);
    }
}
```

* U. S. GOVERNMENT PRINTING OFFICE: 1990--761-051/20139